

第一章：理解网络编程和套接字

理解网络编程和套接字

linux文件操作

第二章：套接字类型与协议设置

套接字协议及其数据传输特性

协议族

套接字类型Type

套接字类型1：面向连接的套接字 (SOCK_STREAM) **TCP**

面向消息的套接字(SOCK_DGRAM) **UDP**

协议的最终选择

第三章：地址族与数据序列

分配给套接字的IP地址和端口号

网络地址

网络地址区分方法

用于区分套接字的端口号

构建结构体来表示IPv4地址

结构体 `sockaddr_in` 的成员分析

网络字节序与地址变换

字节序与网络字节序

网络地址的初始化与分配

将字符串信息转换为网络字节序的整数型

网络地址初始化

客户端地址信息初始化

`INADDR_ANY`

向套接字分配网络地址

问题探讨

IP地址族IPv4和IPv6有何区别？在何种背景下诞生了IPv6？

套接字地址分为IP地址和端口号，为什么需要IP地址和端口号？

请是大端序、小端序、网络字节序，并说明为何需要网络字节序

怎样表示回送地址？其含义是什么？如果向会送地址传输数据会发生什么情况？

第四章：基于TCP的服务器端客户端（1）

TCP/IP协议

TCP服务器端的默认函数调用顺序

TCP客户端的默认函数调用顺序

基于TCP的服务器端、客户端的函数调用关系

实现迭代服务器端/客户端

实现迭代服务器端

迭代回声服务器端、客户端

问题探讨

第五章：基于TCP的服务器端客户端（2）

回声客户端的完美实现

如果问题不在于回声客户端：定义应用层协议

计算器服务端、客户端

TCP原理

TCP套接字中的I/O缓冲

问题探讨

第六章：基于UDP的服务端、客户端

UDP协议的有效性

实现基于UDP的服务器端、客户端

 基于UDP的数据I/O函数

 UDP客户端套接字的地址分配

 UDP的数据传输特性和调用connect函数

 已连接UDP套接字与未连接UDP套接字

 创建已连接UDP套接字

问题探讨

优雅地断开套接字连接

 基于TCP的半关闭

 套接字和流

 针对优雅断开的shutdown函数

 为什么要半关闭

域名及网络地址

 常用的指令

 利用域名获取IP地址 `gethostbyname`

 利用IP地址获取域名 `gethostbyaddr`

套接字的多种可选项

`getsockopt`和`setsockopt`

`SO_SNDBUF` & `SO_RCVBUF`

`SO_REUSEADDR`

 发生地址分配错误

 解决方案：地址再分配机制

`TCP_NODELAY`

 Nagle算法（应用于TCP层）

 禁用Nagle算法

多进程服务器

 并发服务器端的实现方法

 多进程服务器

 理解进程

 通过程序创建进程的方法

 进程和僵尸进程

 产生僵尸进程的原因

 销毁僵尸进程1：利用`wait`函数

 通过`waitpid`函数销毁僵尸进程

 信号处理

 信号与函数

`alarm`函数

 利用`sigaction`函数进行信号处理

 利用信号处理技术消灭僵尸进程

实现并发服务器

分割TCP的I/O程序

问题探讨

进程间通信

进程间通信方式1：通过管道实现进程间通信

通过管道进行进程间双向通信

问题探讨

I/O复用

基于I/O复用的服务器端

理解select函数并实现服务器端

设置文件描述符

设置监视范围及超时

基于I/O复用的回声服务端

问题探究

多种I/O函数

linux中的send和recv函数

MSG_OOB：发送紧急消息

TCP紧急模式

检查输入缓冲

readv和writev函数

问题探讨

多播和广播

多播的数据传输方式及流量方面的优点

路由 (routing) 和TTL (Time to Live, 生存时间) 以及加入组的方法

实现多播Sender和Receiver

广播

广播的理解及实现方法

问题探讨

套接字和标准I/O

使用标准I/O函数的两个优点

标准I/O函数的几个缺点

使用标准I/O函数

基于套接字的标准I/O函数使用

问题探讨

关于I/O流分离的其他内容

分离I/O

文件描述符的复制和半关闭

复制文件描述符

复制文件描述符后“流”的分离

问题探讨

优于select的epoll

基于select的I/O复用技术速度慢的原因

select的缺点

select函数的优点

实现epoll时必要的函数和结构体

select VS epoll

epoll_create

epoll_ctl

epoll_wait

基于epoll的回声服务器端

条件触发(Level Trigger)和边缘触发(Edge Trigger)

实现边缘触发的回声服务器端

边缘触发的服务器端实现中必知的两点

问题探讨

多线程服务器端的实现

线程和进程的差异

线程创建及运行

线程的创建和执行流程

通过 `pthread_join` 函数控制线程的执行流

可在临界区内调用的函数

工作线程模型

线程存在的问题和临界区

多个线程访问同一变量

临界区位置

线程同步

互斥量 (锁机制)

信号量

线程的销毁和多线程并发服务器端的实现

销毁线程的3种方法

多线程并发服务器端的实现

多线程并发客户端实现

问题探讨

总结

关于int main中的传入参数

socket的read和write

write函数

读函数read

fread()和fwrite()

readv和writev函数

TCP三次握手

ACK增量

TCP四次挥手：断开套接字的连接

函数指针

sigaction的信号

Nagle算法

epoll的本质

课后参考

1. 第一章：理解网络编程和套接字

1.1 理解网络编程和套接字

- 第一步：调用socket函数创建套接字。
- 第二步：调用bind函数分配IP地址和端口号。
- 第三步：调用listen函数转为可接收请求状态。
- 第四步：调用accept函数受理连接请求。

```
53. {  
54.     fputs(message, stderr);  
55.     fputc('\n', stderr);  
56.     exit(1);  
57. }  
26.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);  
35.     if(bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)  
38.         if(listen(serv_sock, 5)==-1)  
42.             clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);  
46.             write(clnt_sock, message, sizeof(message));
```

代码说明

- 第26行：调用socket函数创建套接字。
- 第35行：调用bind函数分配IP地址和端口号。
- 第38行：调用listen函数将套接字转为可接收连接状态。
- 第42行：调用accept函数受理连接请求。如果在没有连接请求的情况下调用该函数，则不会返回，直到有连接请求为止。
- 第46行：稍后将要介绍的write函数用于传输数据，若程序经过第42行代码执行到本行，则说明已经有了连接请求。

服务器端创建的套接字又称为**服务器端套接字或监听套接字**

客户端程序只有：

- 调用socket函数创建套接字
- 调用connect函数向服务器端发送链接请求两个步骤

8

```
22.     sock=socket(PF_INET, SOCK_STREAM, 0);  
31.     if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)  
32.         error_handling("connect() error!");
```

代码说明

- 第22行：创建套接字，但此时套接字并不马上分为服务器端和客户端。如果紧接着调用bind、listen函数，将成为服务器端套接字；如果调用connect函数，将成为客户端套接字。
- 第31行：调用connect函数向服务器端发送连接请求。

1.2 linux文件操作

表1-2 文件打开模式

1

打开模式	含 义
O_CREAT	必要时创建文件
O_TRUNC	删除全部现有数据
O_APPEND	维持现有数据，保存到其后面
O_RDONLY	只读打开
O_WRONLY	只写打开
O_RDWR	读写打开

对于多个文件打开模式，用OR运算符组合传递

1.5 习题

- (1) 套接字在网络编程中的作用是什么？为何称它为套接字？
- (2) 在服务器端创建套接字后，会依次调用listen函数和accept函数。请比较并说明二者作用。
- (3) Linux中，对套接字数据进行I/O时可以直接使用文件I/O相关函数；而在Windows中则不可以。原因为何？
- (4) 创建套接字后一般会给它分配地址，为什么？为了完成地址分配需要调用哪个函数？

1.5 习题 25

1

- (5) Linux中的文件描述符与Windows的句柄实际上非常类似。请以套接字为对象说明它们的含义。
- (6) 底层文件I/O函数与ANSI标准定义的文件I/O函数之间有何区别？
- (7) 参考本书给出的示例low_open.c和low_read.c，分别利用底层文件I/O和ANSI标准I/O编写文件复制程序。可任意指定复制程序的使用方法。

1. 套接字是网络数据传输用的软件设备，为了与远程计算机进行数据传输，需要连接到因特网，而编程中的“套接字”就是用来连接该网络的工具。本身就具有连接的意义
2. listen：将套接字转化成可接收连接的状态，accept：接受处理连接请求

2. 第二章：套接字类型与协议设置

2.1 套接字协议及其数据传输特性

创建套接字：

套接字中实际采用的最终协议信息是通过 `socket` 函数的第三个参数传递的，在指定的协议族范围内通过第一个参数决定第三个参数

```
1 #include <sys/socket.h>
2
3 int socket(int domain, int type, int protocol);
4 //domain: 套接字中使用的协议族信息
5 //type: 套接字中数据传输类型信息
6 //Protocol: 计算机间通信中使用的协议信息
```

2.1.1 协议族

表2-1 头文件sys/socket.h中声明的协议族

名 称	协 议 族
PF_INET	IPv4互联网协议族
PF_INET6	IPv6互联网协议族
PF_LOCAL	本地通信的UNIX协议族
PF_PACKET	底层套接字的协议族
PF_IPX	IPX Novell协议族

2.1.2 套接字类型Type

套接字类型指的是套接字的数据传输方式，通过socket的第二个参数传递。只有这样才能决定创建的套接字的数据传输方式。

因为socket的第一个参数 **PF_INET** 协议族中也存在多种数据传输方式

2.1.2.1 套接字类型1：面向连接的套接字 (SOCK_STREAM) **TCP**

如果向socket函数的第二个参数传递SOCK_STREAM，将创建面向连接的套接字。

右图的数据（糖果）传输方式特征整理如下。

面向连接的套接字特点：

- 传输过程中数据不会消失。
- 按序传输数据。
- 传输的数据不存在数据边界 (Boundary)。

知识补给站

套接字缓冲已满是否意味着数据丢失

之前讲过，为了接收数据，套接字内部有一个由字节数组构成的缓冲。如果这个缓冲被接收的数据填满会发生什么事情？之后传递的数据是否会丢失？

首先调用read函数从缓冲读取部分数据，因此，缓冲并不总是满的。但如果read函数读取速度比接收数据的速度慢，则缓冲有可能被填满。此时套接字无法再接收数据，但即使这样也不会发生数据丢失，因为传输端套接字将停止传输。也就是说，面向连接的套接字会根据接收端的状态传输数据，如果传输出错还会提供重传服务。因此，面向连接的套接字除特殊情况外不会发生数据丢失。**TCP协议**

面向连接的套接字特点：

1. 套接字必须一一对应
2. 可靠的、按序传递的、基于字节的面向连接的数据传输方式的套接字
3. 传输的数据不存在数据边界

2.1.2.2 面向消息的套接字(SOCK_DGRAM) **UDP**

面向消息的套接字的特点：

- 强调快速传输而非传输顺序。
- 传输的数据可能丢失也可能损毁。
- 传输的数据有数据边界。 对于大型数据，需要分批发送
- 限制每次传输的数据大小。



总结：不可靠的，不按顺序传递的，以数据的高速传输为目的的套接字

2.1.3 协议的最终选择

socket的创建取决于三个参数：协议族，数据传输类型，计算机通信协议

通常情况下，只需要前两个参数就可以创建所需的套接字，所以大部分情况下第三个参数可以传值 0

当“同一协议族中存在多个数据传输方式相同的协议”时，即数据传输方式相同，但 协议不同，此时需要通过第三个参数具体指定协议信息

IPv4与网络地址系统相关，关于这一点将给出单独说明，目前只需记住：本书是基于IPv4展开的。参数PF_INET指IPv4网络协议族，SOCK_STREAM是面向连接的数据传输。满足这2个条件的协议只有 IPPROTO_TCP，因此可以如下调用socket函数创建套接字，这种套接字称为TCP套接字。

```
int tcp_socket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

下面创建满足如下要求的套接字：

“IPv4协议族中面向消息的套接字”

SOCK_DGRAM指的是面向消息的数据传输方式，满足上述条件的协议只有 IPPROTO_UDP。因此，可以如下调用socket函数创建套接字，这种套接字称为UDP套接字。

```
int udp_socket = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

(1) 什么是协议? 在收发数据中定义协议有何意义? 协议就是用来规定通信双方数据传输格式、方式的标准

(2) 面向连接的TCP套接字传输特性有3点, 请分别说明。一对一、有链接的、可靠的

(3) 下列哪些是面向消息的套接字的特性?

- a. 传输数据可能丢失 ✓
- b. 没有数据边界 (Boundary) ✗
- c. 以快速传递为目标 ✓ ✗
- d. 不限制每次传递数据的大小 ✗
- e. 与面向连接的套接字不同, 不存在连接的概念 ✓

(4) 下列数据适合用哪类套接字传输? 并给出原因。

- a. 演唱会现场直播的多媒体数据 (UDP)
- b. 某人压缩过的文本文件 (TCP)
- c. 网上银行用户与银行之间的数据传递 (TCP)

(5) 何种类型的套接字不存在数据边界? 这类套接字接收数据时需要注意什么?

~~连接~~ (6) ~~tcp_server.c~~ 和 ~~tcp_client.c~~ 中需多次调用 `read` 函数读取服务器端调用 1 次 `write` 函数传递的字符串。更改程序, 使服务器端多次调用 (次数自拟) `write` 函数传输数据, 客户端调用 1 次 `read` 函数进行读取。为达到这一目的, 客户端需延迟调用 `read` 函数, 因为客户端要等待服务器端传输所有数据。Windows 和 Linux 都通过下列代码延迟 `read` 或 `recv` 函数的调用。

```
for(i=0; i<3000; i++)
    printf("Wait time %d \n", i);
```

让CPU执行多余任务以延迟代码运行的方式称为“Busy Waiting”。使用得当即可推迟函数调用。

3. 第三章：地址族与数据序列

套接字的创建相当于只安装了电话机, 而通过地址族等才是给套接字分配IP地址和端口号

数据传输的目的地址应同时包含: 1. IP地址 2. 端口号

3.1 分配给套接字的IP地址和端口号

1. IP(Internet Protocol)网络协议: 为了收发网络数据而分配给计算机的值

2. 端口号: 并非赋予计算机的值, 而是为了区分程序中创建的套接字而分配给套接字的序号

3.1.1 网络地址

为了使计算机连接到网络并收发数据, 必须向其分配IP地址。

- IPv4, 4字节地址族
- IPv6, 16字节地址族



图3-1 IPv4地址族

网络地址（网络ID）是为区分网络而设置的一部分IP地址。假设向WWW.SEMI.COM公司传输数据，该公司内部构建了局域网，把所有计算机连接起来。因此，首先应向SEMI.COM网络传输数据，也就是说，并非一开始就浏览所有4字节IP地址，进而找到目标主机；而是仅浏览4字节IP地址的网络地址，先把数据传到SEMI.COM的网络。SEMI.COM网络（构成网络的路由器）接收到数据后，浏览传输数据的主机地址（主机ID）并将数据传给目标计算机。图3-2展示了数据传输过程。

公网地址
路由转发
ARP

3.1.2 网络地址区分方法

+ 网络地址分类与主机地址边界

只需通过IP地址的第一个字节即可判断网络地址占用的字节数，因为我们根据IP地址的边界区分网络地址，如下所示。

- A类地址的首字节范围：0~127
- B类地址的首字节范围：128~191
- C类地址的首字节范围：192~223

A类：1字节网络ID，3字节主机ID
B类：2字节网络ID，2字节主机ID
C类：3字节网络ID，1字节主机ID
D类：多播IP地址

还有如下这种表述方式。

- A类地址的首位以0开始
- B类地址的前2位以10开始
- C类地址的前3位以110开始

正因如此，通过套接字收发数据时，数据传到网络后即可轻松找到正确的主机。

3.1.3 用于区分套接字的端口号

计算机中一般配有NIC (network interface card, 网络接口卡) 数据传输设备，通过NIC像计算机内部传输数据

端口号是由16位构成，可分配的端口号范围是0-65535,但是0-1023是知名端口，一般分配给特定的应用程序。

虽然端口号不能重复，但是TCP套接字和UDP套接字不会共用端口号。所以允许重复。

即：如果某tcp套接字使用9190端口号，则其他TCP套接字就无法使用该端口号，但UDP套接字可以使用

3.1.4 构建结构体来表示IPv4地址

- 问题1：“采用哪一种地址族？”
- 答案1：“基于IPv4的地址族。”

- 问题2：“IP地址是多少？”
- 答案2：“211.204.214.76。”

40 第3章 地址族与数据序列

- 问题3：“端口号是多少？”
- 答案3：“2048。”

```
1 struct sockaddr_in
2 {
3     sa_family_t sin_family; //地址族
4     uint16_t sin_port; //16位TCP/UDP端口号
5     struct in_addr sin_addr; //32位IP地址
6     char sin_zero[8]; //为了使结构体sockaddr_in的大小与
7     //sockaddr结构体保持一致插入的成员，必须填充为0
8 };
9
10 struct in_addr{
11     In_addr_t s_addr; //32位IPv4地址
12 };
13 //struct sockaddr定义
14 struct sockaddr
15 {
16     sa_family_t sin_family; //地址族
17     char sa_data[14]; //地址信息
18 };
19 /*
20 此结构体成员sa_data保存的地址信息中，需要包含IP地址和端口号。剩余
部分应该填充为0
```

```

21 在这也是bind函数要求的，但这对于包含地址信息来讲非常麻烦，因此有了
22 新的结构体sockaddr_in
23 通过将sockaddr_in的地址再转换为sockaddr型的结构体变量，传入
24 bind
25 */
26 //bind示例
27 if(bind(serv_sock, (struct sockaddr*)&serv_addr,
28         sizeof(serv_addr)) == -1)
29     error_handling("bind() error");

```

(Portable Operating System Interface，可移植操作系统接口)。POSIX是为UNIX系列操作系统设立的标准，它定义了一些其他数据类型，如表3-1所示。

表3-1 POSIX中定义的数据类型

数据类型名称	数据类型说明	声明的头文件
int8_t	signed 8-bit int	
uint8_t	unsigned 8-bit int (unsigned char)	
int16_t	signed 16-bit int	
uint16_t	unsigned 16-bit int(unsigned short)	sys/types.h
int32_t	signed 32-bit int	
uint32_t	unsigned 32-bit int(unsigned long)	
sa_family_t	地址族 (address family)	
socklen_t	长度 (length of struct)	sys/socket.h
in_addr_t	IP地址，声明为uint32_t	
in_port_t	端口号，声明为uint16_t	netinet/in.h

3.1.4.1 结构体sockaddr_in的成员分析

成员sin_family

每种协议族适用的地址族均不同。比如，IPv4使用4字节地址族，IPv6使用16字节地址族。可以参考表3-2保存sin_family地址信息。

表3-2 地址族

地址族 (Address Family)	含 义
AF_INET	IPv4网络协议中使用的地址族
AF_INET6	IPv6网络协议中使用的地址族
AF_LOCAL	本地通信中采用的UNIX协议的地址族

AF_LOCAL只是为了说明具有多种地址族而添加的，希望各位不要感到太突然。

成员sin_port

该成员保存16位端口号，重点在于，它以网络字节序保存(关于这一点稍后将给出详细说明)。

成员sin_addr

该成员保存32位IP地址信息，且也以网络字节序保存。为理解好该成员，应同时观察结构体in_addr。但结构体in_addr声明为uint32_t，因此只需当作32位整数型即可。

成员sin_zero

无特殊含义。只是为使结构体sockaddr_in的大小与sockaddr结构体保持一致而插入的成员。必需填充为0，否则无法得到想要的结果。后面会另外讲解sockaddr。

从之前介绍的代码也可看出，sockaddr_in结构体变量地址值将以下方式传递给bind函数。稍后将给出关于bind函数的详细说明，希望各位重点关注参数传递和类型转换部分的代码。

```
struct sockaddr_in serv_addr;
...
if(bind(serv_sock, (struct sockaddr * ) &serv_addr, sizeof(serv_addr)) == -1)
    error_handling("bind() error");
```

知识补给站

sin_family

sockaddr_in是保存IPv4地址信息的结构体。那为何还需要通过sin_family单独指定地址族信息呢？这与之前讲过的sockaddr结构体有关。结构体sockaddr并非只为IPv4设计，这从保存地址信息的数组sa_data长度为14字节也可看出。因此，结构体sockaddr要求在sin_family中指定地址族信息。为了与sockaddr保持一致，sockaddr_in结构体中也有地址族信息。
4

3.2 网络字节序与地址变换

3.2.1 字节序与网络字节序

CPU向内存保存数据的方式有两种：

- 大端序：高位字节存放到低位地址
- 小端序：高位字节存放到高位地址
- 目前主流的CPU以小端序方式保存数据

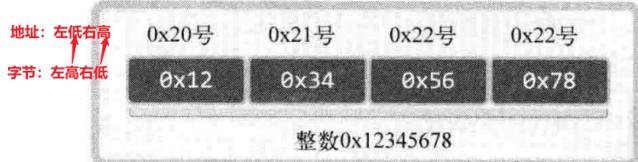


图3-4 大端序字节表示

整数0x12345678中，0x12是最高位字节，0x78是最低位字节。因此，大端序中先保存最高位字节0x12（最高位字节0x12存放到低位地址）。小端序保存方式如图3-5所示。

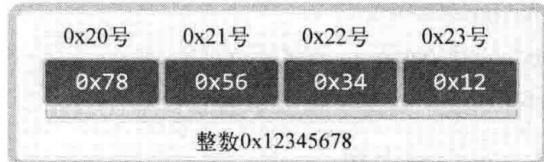


图3-5 小端序字节表示

网络字节序：为了解决接受方和发送方可能存在的字节序不同导致的数据传输问题 (统一为大端序)

- `unsigned short htons(unsigned short);`
- `unsigned short ntohs(unsigned short);`
- `unsigned long htonl(unsigned long);`
- `unsined long ntohl(unsigned long),`

通过函数名应该能掌握其功能，只需了解以下细节。

- htons中的h代表主机（host）字节序。
- htons中的n代表网络（network）字节序。

另外，s指的是short，l指的是long（Linux中long类型占用4个字节，这很关键）。因此，htons是h、to、n、s的组合，也可以解释为“把short型数据从主机字节序转化为网络字节序”。

再举个例子，ntohs可以解释为“把short型数据从网络字节序转化为主机字节序”。

通常，以s作为后缀的函数中，s代表2个字节short，因此用于端口号转换。以l作为后缀的函数中，l代表4个字节，因此用于IP地址转换。另外，有些读者可能有如下疑问：

```

1. #include <stdio.h>
2. #include <arpa/inet.h>
3.
4. int main(int argc, char *argv[])
5. {
6.     unsigned short host_port=0x1234;
7.     unsigned short net_port;
8.     unsigned long host_addr=0x12345678;
9.     unsigned long net_addr;
10.
11.    net_port=htons(host_port); host_port to network_Port
12.    net_addr=htonl(host_addr); host_addr to network_addr

```

```

1 #include <stdio.h>
2 #include <arpa/inet.h>
3
4 int main(int argc, char *argv[])
5 {

```

```

6     unsigned short host_port = 0x1234;
7     unsigned short net_port;
8     unsigned long host_addr = 0x12345678;
9     unsigned long net_addr;
10
11    net_port = htons(host_port);
12    net_addr = htonl(host_addr);
13
14    printf("Host ordered port: %#x \n", host_port);
//0x1234
15    printf("Network ordered prot: %#x \n", net_port);
16    printf("Host ordered address: %#lx \n", host_addr);
17    printf("Network ordered address: %#lx \n",
net_addr);
18
19    return 0;
20 }
```

```

[xiaotang@VM-0-5-centos src]$ vim endian_conv.c
[xiaotang@VM-0-5-centos src]$ gcc endian_conv.c -o conv
[xiaotang@VM-0-5-centos src]$ ./conv
Host ordered port: 0x1234 小端序cpu排列
Network ordered prot: 0x3412 network_port, network_addr都是采用大端
Host ordered address: 0x12345678
Network ordered address: 0x78563412
[1]+ 1199 Stopped                 ./conv
```

3.3 网络地址的初始化与分配

3.3.1 将字符串信息转换为网络字节序的整数型

`sockaddr_in` 中保存地址信息的成员是32位整型，为了帮助我们将字符串形式的ip地址转换为32位整数型数据，因此需要通过 `inet_addr` 函数在转换类型的同时进行网络字节序转换

```

1 #include <arpa/inet.h>
2
3 in_addr_t inet_addr(const char* string); //成功时返回32位
大端序整数型值，失败时返回INADDR_NONE
4
5 int inet_aton(const char* string, struct in_addr
*addr); //成功时返回1，失败时返回0
6 //string:含有需要转换的IP地址信息的字符串地址
7 //addr: 将保存转换结果的in_addr结构体变量的地址值
8
```

```
9 //测试代码
10 #include <stdio.h>
11 #include <arpa/inet.h>
12
13 int main(int argc, char *argv[])
14 {
15     char *addr1 = "1.2.3.4"; //小端为0x01020304
16     char *addr2 = "1.2.3.256";
17
18     unsigned long conv_addr = inet_addr(addr1);
19
20     if(conv_addr == INADDR_NONE) printf("Error
21 occurred!\n");
22     else printf("Network ordered integer addr: %#lx
23 \n", conv_addr);
24
25     conv_addr = inet_addr(addr2);
26     if(conv_addr == INADDR_NONE) printf("Error
27 occurred!\n");
28     else printf("Network ordered integer addr: %#lx
29 \n", conv_addr);
30
31     return 0;
32 }
33 //显示结果为
```

```
[xiaotang@VM-0-5-centos src]$ ./inet_addr
Network ordered integer addr: 0x4030201 大端
Error occurred!
```

实际编程中若要调用inet_addr函数，需将转换后的IP地址信息代入sockaddr_in结构体中声明的in_addr结构体变量。而inet_aton函数则不需此过程。原因在于，若传递in_addr结构体变量地址值，函数会自动把结果填入该结构体变量。通过示例了解inet_aton函数调用过程。

```

❖ inet_aton.c

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <arpa/inet.h>
4. void error_handling(char *message);
5.
6. int main(int argc, char *argv[])
7. {
8.     char *addr="127.232.124.79";
9.     struct sockaddr_in addr_inet;
10.
11.    if(!inet_aton(addr, &addr_inet.sin_addr))
12.        error_handling("Conversion error");
13.    else
14.        printf("Network ordered integer addr: %#x \n",
15.               addr_inet.sin_addr.s_addr);
16.
17.    return 0;
18.

```

```

struct sockaddr_in
{
    sa_family_t sin_family; //地址族
    uint16_t sin_port; //16位TCP/UDP端口号
    struct in_addr sin_addr; //32位IP地址
    char sin_zero[8];
};

struct in_addr(
    in_addr_t s_addr; //32位IPv4地址
);

//struct sockaddr
struct sockaddr
{
    sa_family_t sin_family; //地址族
    char sa_data[14]; //地址信息
};

```

总结：三种字符串和网络字节序转换的函数

```

1 #include <arpa/inet.h>
2
3 in_addr_t inet_addr(const char* string); //成功时返回32位
大端序整数值，失败时返回INADDR_NONE
4
5 int inet_aton(const char* string, struct in_addr
*addr); //成功时返回1，失败时返回0
//string:含有需要转换的IP地址信息的字符串地址
//addr: 将保存转换结果的in_addr结构体变量的地址值
6
7
9 char* inet_ntoa(struct in_addr adr); //将网络字节序整型
IP地址转换成我们熟悉的字符串形式
10 //成功时返回转换的字符串地址值，失败返回-1

```

❖ inet_ntoa.c

```

1. #include <stdio.h>
2. #include <string.h>
3. #include <arpa/inet.h>
4.
5. int main(int argc, char *argv[])
6. {
7.
8.     struct sockaddr_in addr1, addr2;
9.     char *str_ptr;
10.    char str_arr[20];
11.    addr1.sin_addr.s_addr=htonl(0x1020304);
12.    addr2.sin_addr.s_addr=htonl(0x1010101);
13.    str_ptr=inet_ntoa(addr1.sin_addr);
14.    strcpy(str_arr, str_ptr);
15.    printf("Dotted-Decimal notation1: %s \n", str_ptr);
16.
17.    inet_ntoa(addr2.sin_addr);
18.    printf("Dotted-Decimal notation2: %s \n", str_ptr);
19.    printf("Dotted-Decimal notation3: %s \n", str_arr);
20.
21.    return 0;
22.

```

49

主机IP地址转网络地址

网络地址转为IP地址，用char*保存

str_arr:0x1020304对应的ip
str_ptr: 第一次是0x102030
第二次是0x1010101

3.3.2 网络地址初始化

结合前面所学的内容，现在介绍套接字创建过程中常见的网络地址信息初始化方法。

```
struct sockaddr_in addr;
char * serv_ip = "211.217.168.13"; //声明 IP 地址字符串
char * serv_port = "9190"; //声明端口号字符串
memset(&addr, 0, sizeof(addr)); //结构体变量 addr 的所有成员初始化为 0
addr.sin_family = AF_INET; //指定地址族
addr.sin_addr.s_addr = inet_addr(serv_ip); //基于字符串的 IP 地址初始化
addr.sin_port = htons(atoi(serv_port)); //基于字符串的端口号初始化
atof: char* to int
```

上述代码中，`memset`函数将每个字节初始化为同一值：第一个参数为结构体变量`addr`的地址值，即初始化对象为`addr`；第二个参数为0，因此初始化为0；最后一个参数中传入`addr`的长度，

因此`addr`的所有字节均初始化为0。这么做是为了将`sockaddr_in`结构体的成员`sin_zero`初始化为0。另外，`inet_addr`函数把字符串类型的值转换成整型。总之，上述代码利用字符串格式的IP地址和端口号初始化了`sockaddr_in`结构体变量。

另外，代码中对IP地址和端口号进行了硬编码，这并非良策，因为运行环境改变就得更改代码。因此，我们运行示例`main`函数时传入IP地址和端口号。

3.3.3 客户端地址信息初始化

在3.3.2节中的网络地址信息初始化过程主要针对服务器端而非客户端，给套接字分配IP地址和端口号主要是为了让服务器进入`listen`并`accept`的状态

请求方法不同意味着调用的函数也不同。服务器端的准备工作通过`bind`函数完成，而客户端则通过`connect`函数完成。因此，函数调用前需准备的地址值类型也不同。服务器端声明`sockaddr_in`结构体变量，将其初始化为赋予服务器端IP和套接字的端口号，然后调用`bind`函数；而客户端则声明`sockaddr_in`结构体，并初始化为要与之连接的服务器端套接字的IP和端口号，然后调用`connect`函数。

3.3.4 INADDR_ANY

在3.3.2节中，每次创建服务器端套接字都要输入IP地址，因此可以通过如下方式初始化地址信息

```
1 struct sockaddr_in addr;
2 char * serv_port = "9190";
3
4 memset(&addr, 0, sizeof(addr));
5 addr.sin_family = AF_INET; //协议族
6 addr.sin_addr.s_addr = htonl(INADDR_ANY); //利用常数
INADDR_ANY分配服务器端的IP地址
7 addr.sin_port = htons(atoi(serv_port)); //基于字符串的网络
端口初始化
```

与之前方式最大的区别在于，利用常数INADDR ANY分配服务器端的IP地址。若采用这种方式，则可自动获取运行服务器端的计算机IP地址，不必亲自输入。而且，若同一计算机中已分配多个IP地址（多宿主（Multi-homed）计算机，一般路由器属于这一类），则只要端口号一致，就可以从不同IP地址接收数据。因此，服务器端中优先考虑这种方式。而客户端中除非带有一部分服务器端功能，否则不会采用。~~（注：此段文字有误，应为“若同一计算机中已分配多个IP地址（多宿主（Multi-homed）计算机，一般路由器属于这一类），则只要端口号一致，就可以从不同IP地址接收数据。因此，服务器端中优先考虑这种方式。而客户端中除非带有一部分服务器端功能，否则不会采用。”）~~

知识补给站

创建服务器端套接字时需要IP地址的原因

初始化服务器端套接字时应分配所属计算机的IP地址，因为初始化时使用的IP地址非常明确，那为何还要进行IP初始化呢？如前所述，同一计算机中可以分配多个IP地址，实际IP地址的个数与计算机中安装的NIC的数量相等。即使是服务器端套接字，也需要决定应接收哪个IP传来的（哪个NIC传来的）数据。因此，服务器端套接字初始化过程中要求IP地址信息。另外，若只有1个NIC，则直接使用INADDR_ANY。

3.3.5 向套接字分配网络地址

在完成 `sockaddr_in` 结构体的初始化方法后，需要通过 `bind` 函数 负责将初始化的地址信息分配给套接字

```
1 #include <sys/socket.h>
2
3 int bind(int sockfd, struct sockaddr * myaddr,
4          socklen_t addrlen);
5 //sockfd: 要分配地址信息（IP地址和端口号）的套接字文件描述符
6 //myaddr: 存有地址信息的结构体变量地址值
7 //addrlen: 第二个结构体变量的长度
```

下面给出服务器端常见套接字初始化过程。

```
int serv_sock;           // 服务端的套接字
struct sockaddr_in serv_addr; // 地址信息结构体
char * serv_port = "9190"; // 端口地址

/* 创建服务器端套接字（监听套接字） */
serv_sock = socket(PF_INET, SOCK_STREAM, 0);
// 协议族          // 数据传输方式：面向连接的

/* 地址信息初始化 */
memset(&serv_addr, 0, sizeof(serv_addr)); // 初始化结构体，赋值为0
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(atoi(serv_port));

/* 分配地址信息 */
bind(serv_sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
.....               // 类型转换
```

3.4 问题探讨

3.4.1 IP地址族IPv4和IPv6有何区别？在何种背景下诞生了IPv6？

IPv4和IPv6都是IP地址族，只是长度不一样。分别是四个字节和16个字节。

为了解决IP地址消耗的问题

3.4.2 套接字地址分为IP地址和端口号，为什么需要IP地址和端口号？

IP地址可以用来区分网络上计算机，而端口号可以在一台计算上用来区分不同套接字而设置的。

知名端口是：0-1023

3.4.3 请是大端序、小端序、网络字节序，并说明为何需要网络字节序

1. 大端序：计算机cpu向内存写入数据时，高位数据放在低位地址
2. 小端序：
3. 网络字节序：为了解决通信双方因为端序不一致而导致的数据传输问题

3.4.4 怎样表示回送地址？其含义是什么？如果向会送地址传输数据会发生什么情况？

回送地址（127.x.x.x）是本机回送地址（Loopback Address），即主机IP堆栈内部的IP地址，主要用于网络软件测试以及本地机进程间通信，无论什么程序，一旦使用回送地址发送数据，协议软件立即返回之，不进行任何网络传输。属于保留测试地址，不能用，同时网络ID的第一个6位组也不能全置为“0”，全“0”表示本地网络。

4. 第四章：基于TCP的服务器端客户端（1）

根据数据传输方式的不同，基于网络协议的套接字一般分为TCP套接字和UDP套接字，因为TCP套接字是面向连接的，因此又称基于流的套接字（字节流）

4.1 TCP/IP协议

1. 数据链路层
2. 网络层
3. 传输层
4. 应用层：将数据传输路径，数据确认过程都隐藏到套接字内部。这也是网络编程称为套接字编程的原因

4.1.1 TCP服务器端的默认函数调用顺序

1. socket()创建套接字
2. bind()分配套接字地址
3. listen()等待连接请求状态
4. accept()允许连接
5. read()/write() 数据交换传输
6. close()断开链接

+ 进入等待连接请求状态

我们已调用bind函数给套接字分配了地址，接下来就要通过调用listen函数进入等待连接请求状态。只有调用了listen函数，客户端才能进入可发出连接请求的状态。换言之，这时客户端才能调用connect函数（若提前调用将发生错误）。

```
#include <sys/socket.h>

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

→ 成功时返回 0，失败时返回-1。

4

- sock 希望进入等待连接请求状态的套接字文件描述符，传递的描述符套接字参数成为服务器端套接字（监听套接字）。
- backlog 连接请求等待队列（Queue）的长度，若为5，则队列长度为5，表示最多使5个连接请求进入队列。

等待连接请求状态：指客户端请求连接时，受理连接前一直处于等待状态

连接请求等待队列：允许的等待序队列的长度

准备好服务器端套接字和连接请求等待队列后，这种可接受连接请求的状态称为**等待连接请求状态**

+ 受理客户端连接请求

调用listen函数后，若有新的连接请求，则应按序受理。受理请求意味着进入可接受数据的状态。也许各位已经猜到进入这种状态所需部件——当然是套接字！大家可能认为可以使用服务器端套接字，但服务器端套接字是做门卫的。如果在与客户端的数据交换中使用门卫，那谁来守门呢？因此需要另外一个套接字，但没必要亲自创建。下面这个函数将自动创建套接字，并连接到发起请求的客户端。

```
#include <sys/socket.h>

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

→ 成功时返回创建的套接字文件描述符，失败时返回-1。

- sock 服务器套接字的文件描述符。
- addr 保存发起连接请求的客户端地址信息的变量地址值，调用函数后向传递来的地址变量参数填充客户端地址信息。
- addrlen 第二个参数addr结构体的长度，但是存有长度的变量地址。函数调用完成后，该变量即被填入客户端地址长度。

accept函数受理连接请求等待队列中待处理的客户端连接请求。函数调用成功时，accept函数内部将产生用于数据I/O的套接字，并返回其文件描述符。需要强调的是，套接字是自动创建的，并自动与发起连接请求的客户端建立连接。图4-8展示了accept函数调用过程。

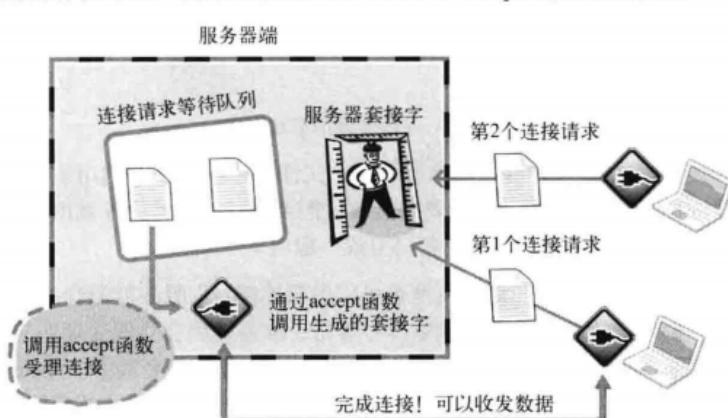


图4-8 受理连接请求状态

4.1.2 TCP客户端的默认函数调用顺序

1. socket() 创建套接字
2. connect()请求连接
3. read() / write() 交换数据
4. close() 断开连接

与服务器端相比，区别就在于“请求连接”，在客户端创建套接字后向服务器端发起的连接请求。但必须在服务器端调用 listen函数后创建请求等待队列，之后客户端才可请求连接

```

1 #include <sys/socket.h>
2
3 int connect(int sock, struct sockaddr * servaddr,
4             socklen_t addrlen);
5 //sock:客户端套接字文件描述符
6 //servaddr: 保存目标服务器端地址信息的变量地址值
7 //addrlen: 以字节为单位传递的地址变量长度

```

客户端在调用connect函数后，发生以下情况之一才会返回：

1. 服务器端接受连接请求（并不意味着服务器端调用accept函数，可以是进入到等待队列）因此，connect函数返回后，并不立即进行数据交换
2. 发生断网等异常情况而中断连接请求

知识补给站 客户端套接字地址信息在哪？

实现服务器端必经过程之一就是给套接字分配IP和端口号，但客户端实现过程中并未出现套接字地址分配，而是创建套接字后立即调用connect函数。难道客户端套接字无需分配IP和端口？当然不是！**网络数据交换必须分配IP和端口。**既然如此，那客户端套接字何时、何地、如何分配地址呢？

- 何时？调用connect函数时。
- 何地？操作系统，更准确地说是在内核中。
- 如何？IP用计算机（主机）的IP，端口随机。

客户端的IP地址和端口在调用connect函数时自动分配，**无需调用标记的bind函数进行分配。**

4.1.3 基于TCP的服务器端、客户端的函数调用关系

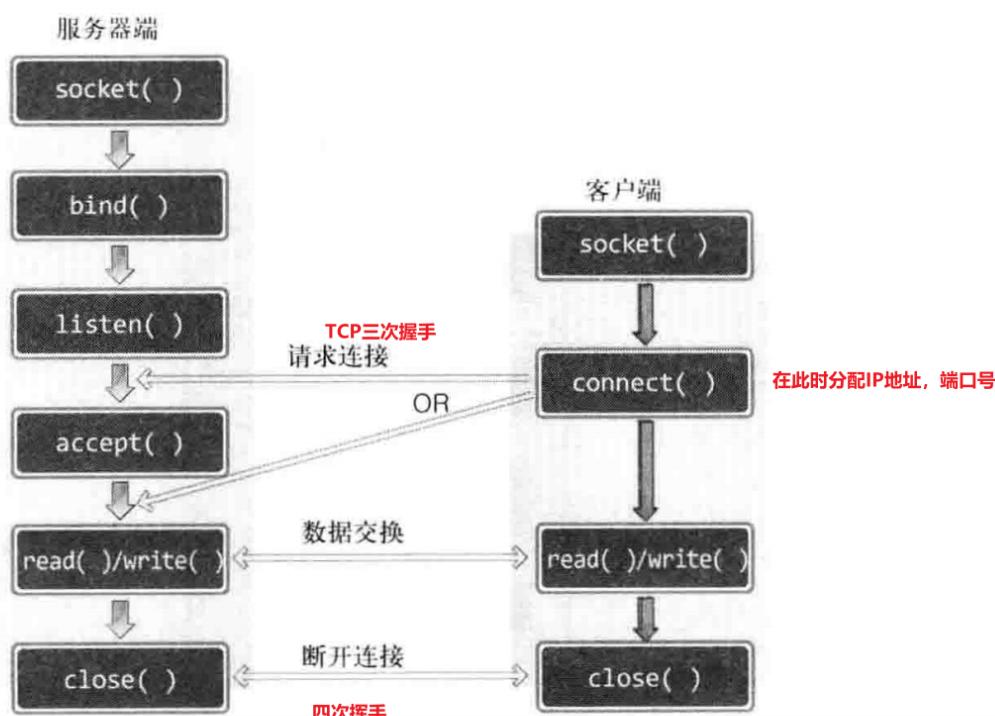


图4-10 函数调用关系

图4-10的总体流程整理如下：服务器端创建套接字后连续调用bind、listen函数进入等待状态，客户端通过调用connect函数发起连接请求。需要注意的是，**客户端只能等到服务器端调用listen函数后才能调connect函数**。同时要清楚，客户端调用connect函数前，服务器端有可能率先调用accept函数。当然，此时服务器端在调用accept函数时进入阻塞（blocking）状态，直到客户端调用connect函数为止。

4.2 实现迭代服务器端/客户端

本节编写回声服务器端/客户端，即服务器端将客户端传输的字符串数据原封不动地传回给客户端

4.2.1 实现迭代服务器端

没有太大意义。但这并非我们想象的服务器端。设置好等待队列的大小后，应向所有客户端提供服务。如果想继续受理后续的客户端连接请求，应怎样扩展代码？**最简单的办法就是插入循环语句反复调用accept函数**，如图4-11所示。

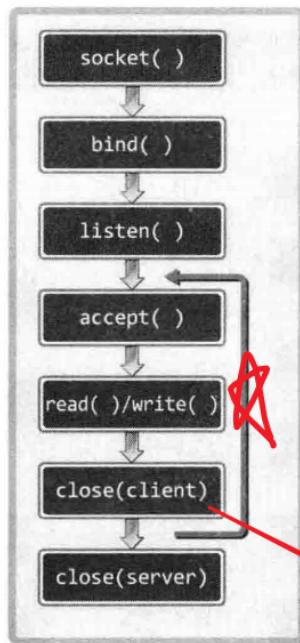


图4-11 迭代服务器端的函数调用顺序

从图4-11可以看出，调用accept函数后，紧接着调用I/O相关的read、write函数，然后调用close函数。这并非针对服务器端套接字，而是**针对accept函数调用时创建的套接字**。

门卫还在，但是换了一批面试官

调用close函数就意味着结束了针对某一客户端的服务。此时如果还想服务于其他客户端，就要重新调用accept函数。

“这算什么呀？又不是银行窗口，好歹也是个服务器端，难道同一时刻只能服务于一个客户端吗？”

是的！同一时刻确实只能服务于一个客户端。将来学完进程和线程后，就可以编写同时服务多个客户端的服务器端了。目前只能做到这一步，虽然很遗憾，但请各位不要心急。**我不着急，我很期待**

4.2.1.1 迭代回声服务器端、客户端

- 服务器端在同一时刻只与一个客户端相连，并提供回声服务。
- 服务器端依次向5个客户端提供服务并退出。
- 客户端接收用户输入的字符串并发送到服务器端。
- 服务器端将接收的字符串数据传回客户端，即“回声”。
- 服务器端与客户端之间的字符串回声一直执行到客户端输入Q为止。

存在的问题：

每次调用write函数都会传递一个字符串，因此这种假设在某种程度上合理。

但是TCP不存在数据边界，而上述客户端是基于TCP的，因此多次调用write函数传递的字符串有可能一次性传递到服务端，此时客户端有可能从服务端收到多个字符串。

因此，还需要考虑服务器端的如下情况：字符串太长则分成两个数据包发送

解决办法：可以提前确定接受数据的大小

❖ echo_server.c

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <unistd.h>
5. #include <arpa/inet.h>
6. #include <sys/socket.h>
7.
8. #define BUF_SIZE 1024
9. void error_handling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     int serv_sock, clnt_sock;
14.     char message[BUF_SIZE];
15.     int str_len, i;
16.
17.     struct sockaddr_in serv_addr, clnt_addr;      定义服务器端和客户端的地址
18.     socklen_t clnt_addr_sz;                      客户端地址长度
19.
20.     if(argc!=2) {
21.         printf("Usage : %s <port>\n", argv[0]);
22.         exit(1);
```

```

23. }
24.
25. serv_sock=socket(PF_INET, SOCK_STREAM, 0); 通过socket生成套接字  
PF_INET, IPv4网络  
SOCK_STREAM, 面向连接的流数据对象
26. if(serv_sock==-1)
27.     error_handling("socket() error");
28. 套接字初始化工作
29. memset(&serv_addr, 0, sizeof(serv_addr));
30. serv_addr.sin_family=AF_INET;
31. serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
32. serv_addr.sin_port=htons(atoi(argv[1]));
33.
34. if(bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
35.     error_handling("bind() error"); 通过bind进行地址分配
36.
37. if(listen(serv_sock, 5)==-1) 进入listen状态
38.     error_handling("listen() error");
39.
40. clnt_addr_sz(sizeof(clnt_addr));
41.

42. for(i=0; i<5; i++) 至多连接五个client
43. {
44.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_sz); accept: 接受客户端的请求，并自动生成I/O套接字
45.     if(clnt_sock==-1)
46.         error_handling("accept() error");
47.     else
48.         printf("Connected client %d \n", i+1);
49.
50.     while((str_len=read(clnt_sock, message, BUF_SIZE))!=0) 对每个client，自动读取并反馈字符串回去
51.         write(clnt_sock, message, str_len);
52.
53.     close(clnt_sock); 关闭当前连接
54. }
55. close(serv_sock); 关闭服务器
56. return 0;
57. }

58.
59. void error_handling(char *message)
60. {
61.     fputs(message, stderr);
62.     fputc('\n', stderr);
63.     exit(1);
64. }

echo_client.c
4. #include <unistd.h>
5. #include <arpa/inet.h>
6. #include <sys/socket.h>
7.
8. #define BUF_SIZE 1024
9. void error_handling(char *message);
10.
11. int main(int argc, char *argv[])
12. {
13.     int sock;
14.     char message[BUF_SIZE];
15.     int str_len;
16.     struct sockaddr_in serv_addr;
17.
18.     if(argc!=3) {
19.         printf("Usage : %s <IP> <port>\n", argv[0]);
20.         exit(1);
21.     }
22.
23.     sock=socket(PF_INET, SOCK_STREAM, 0); socket创建套接字
24.     if(sock==-1)
25.         error_handling("socket() error");
26.

```

```

27.     memset(&serv_addr, 0, sizeof(serv_addr)); 直接初始化套接字
28.     serv_addr.sin_family=AF_INET;
29.     serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
30.     serv_addr.sin_port=htons(atoi(argv[2]));
31.
32.     if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
33.         error_handling("connect() error!"); 通过connect进行套接字的IP, 端口号初始化。
34.     else 而不需要bind函数
35.         puts("Connected.....");
36.
37.     while(1)
38.     {
39.         fputs("Input message(Q to quit): ", stdout);
40.         fgets(message, BUF_SIZE, stdin);
41.
42.         if(!strcmp(message, "q\n") || !strcmp(message, "Q\n"))
43.             break;
44.
45.         write(sock, message, strlen(message));
46.         str_len=read(sock, message, BUF_SIZE-1);
47.         message[str_len]=0;
48.         printf("Message from server: %s", message);
49.     }

```

4.3 问题探讨

1. 客户端调用connect函数向服务器端发送连接请求，服务器端调用哪个函数后，客户端可以调用connect函数

服务器端必须在socket(), bind(), listen()之后才能够接受connect()请求

2. 什么时候创建请求等待队列？它有何作用？与accept有什么关系？

在服务器端套接字繁忙的时候，会创建一个请求等待序列，对于新的连接请求，按序处理

因此，客户端的connect成功，不一定就真正被服务器端accept，可能只是被listen的

accept函数内部自动产生用于数据I/O的套接字，并自动与发起连接请求的客户端建立连接

3. 客户端中为何不需要调用bind函数分配地址？如果不调用bind函数，那么何时、如何向套接字分配IP地址和端口号？

客户端也需要为套接字分配IP和端口，只是在调用connect函数的时候自动分配。无需调用标记的bind函数进行分配

何时？调用connect函数时

何地？系统内核中

如何？IP用计算机主机IP，端口随机

5. 第五章：基于TCP的服务器端客户端（2）

5.1 回声客户端的完美实现

在第四章分析得到的回声客户端存在的问题：

每次调用write函数都会传递一个字符串，因此这种假设在某种程度上合理。

但是 **TCP不存在数据边界**，而上述客户端是基于TCP的，因此多次调用write函数传递的字符串有可能一次性传递到服务端，此时客户端有可能从服务端收到多个字符串。

因此，还需要考虑服务器端的如下情况：字符串太长则分成两个数据包发送

```
while(1)
{
    fputs("Input message(Q to quit): ", stdout);
    fgets(message, BUF_SIZE, stdin);
    ...
    write(sock, message, strlen(message));
    str_len = read(sock, message, BUF_SIZE - 1);
    message[str_len] = 0;
    printf("Message from server: %s", message);
}
```

大家现在理解了吧？回声客户端传输的是字符串，而且是通过调用write函数一次性发送的。之后还调用一次read函数，期待着接收自己传输的字符串。这就是问题所在。

“既然回声客户端会收到所有字符串数据，是否只需多等一会儿？过一段时间后再调用read函数是否可以一次性读取所有字符串数据？”

的确，过一段时间后即可接收，但需要等多久？要等10分钟吗？这不符合常理，理想的客户端应在收到字符串数据时立即读取并输出。

```
36.     while(1)
37.     {
38.         fputs("Input message(Q to quit): ", stdout);
39.         fgets(message, BUF_SIZE, stdin);
40.         if(!strcmp(message, "q\n") || !strcmp(message, "Q\n"))
41.             break;
42.
43.         str_len=write(sock, message, strlen(message)); 通过记录发送的字符串长度
44.
45.         recv_len=0;
46.         while(recv_len<str_len) 判断当前接受的字符串长度是否是发送的字符串长度
47.         {
48.             recv_cnt=read(sock, &message[recv_len], BUF_SIZE-1);
49.             if(recv_cnt== -1)
50.                 error_handling("read() error!");
51.             recv_len+=recv_cnt;
52.         }
53.         message[recv_len]=0;
54.         printf("Message from server: %s", message);
55.     }
```

5.2 如果问题不在于回声客户端：定义应用层协议

回声客户端可以提前知道要接收的数据长度，但是在大多数情况下，这不太可能。

因此，需要通过**定义应用层协议**：在收发数据过程中需要定好规则以表示数据的**边界**，或提前告知收发数据的**大小**

5.2.1 计算器服务端、客户端

我编写程序前设计了**如下应用层协议**，但这只是为实现程序而设计的最低协议，实际的应用程序实现中需要的协议更详细、准确。

- 客户端连接到服务器端后以1字节整数形式传递待算数字个数。
- 客户端向服务器端传递的每个整型数据占用4字节。
- 传递整型数据后接着传递运算符。运算符信息占用1字节。
- 选择字符+、-、*之一传递。
- 服务器端以4字节整型向客户端传回运算结果。
- 客户端得到运算结果后终止与服务器端的连接。

5.3 TCP原理

5.3.1 TCP套接字中的I/O缓冲

实际上，**write函数调用后并非立即传输数据，read函数调用后也并非马上接收数据**。更准确地说，如图5-2所示，**write函数调用瞬间，数据将移至输出缓冲；read函数调用瞬间，从输入缓冲读取数据**。

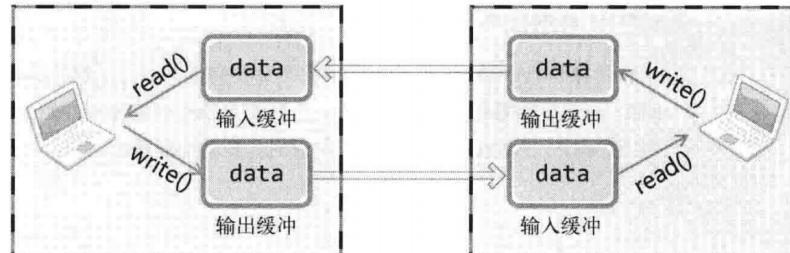


图5-2 TCP套接字的I/O缓冲

如图5-2所示，调用**write**函数时，数据将移到输出缓冲，在适当的时候（不管是分别传送还是一次性传送）传向对方的输入缓冲。这时对方将调用**read**函数从输入缓冲读取数据。这些I/O缓冲特性可整理如下。

- I/O缓冲在每个TCP套接字中**单独存在**。
- I/O缓冲在创建套接字时**自动生成**。
- 即使关闭套接字也会继续传递**输出缓冲中遗留的数据**。
- 关闭套接字将**丢失输入缓冲中的数据**。

也就是说，根本不会发生这类问题，因为TCP会控制数据流。TCP中有滑动窗口（Sliding Window）协议，用对话方式呈现如下。

TCP流量控制

- 套接字A：“你好，最多可以向我传递50字节。”
- 套接字B：“OK！”

- 套接字A：“我腾出了20字节的空间，最多可以收70字节。”
- 套接字B：“OK！”

5.4 问题探讨

1. 请说明TCP套接字连接设置的三次握手过程，尤其是3次数据交换过程每次收发的数据内容

首先：client向服务器端发送SEQ，并要求server返回一定格式的ACK

Server：SEQ：告诉客户端应该以什么样的形式ACK，并且附加一个ACK对client的SEQ确认

Client：发送server需要的SEQ，并对server的SEQ进行ACK

2. TCP是可靠的数据传输协议，但在通过网络通信的过程中可能丢失数据，请通过ACK和SEQ说明TCP通过何种机制保证丢失数据的可靠传输
3. 对方主机的输入缓冲剩余50字节空间时，若本方主机通过write函数请求传输70字节，请问tcp如何处理这种情况？

通过流量控制

(6) 创建收发文件的服务器端/客户端，实现顺序如下。

- 客户端接受用户输入的传输文件名。
- 客户端请求服务器端传输该文件名所指文件。
- 如果指定文件存在，服务器端就将其发送给客户端；反之，则断开连接。

6. 第六章：基于UDP的服务端、客户端

TCP在不可靠的IP层进行流控制，进而实现提供可靠的数据传输服务

流控制是区分UDP和TCP的最重要标志

6.1 UDP协议的有效性

用UDP更有效。讲解前希望各位明白，UDP也具有一定的可靠性。网络传输特性导致信息丢失频发，可若要传递压缩文件（发送1万个数据包时，只要丢失1个就会产生问题），则必须使用TCP，因为压缩文件只要丢失一部分就很难解压。但通过网络实时传输视频或音频时的情况有所不同。对于多媒体数据而言，丢失一部分也没有太大问题，这只会引起短暂的画面抖动，或出现细微的杂音。但因为需要提供实时服务，速度就成为非常重要的因素。因此，第5章的流控制就显得有些多余，此时需要考虑使用UDP。但UDP并非每次都快于TCP，TCP比UDP慢的原因通常有以下两点。

- 收发数据前后进行的连接设置及清除过程。①
- 收发数据过程中为保证可靠性而添加的流控制②

如果收发的数据量小但需要频繁连接时，UDP比TCP更高效。有机会的话，希望各位深入学习TCP/IP协议的内部构造。C语言程序员懂得计算机结构和操作系统知识就能写出更好的程序，同样，网络程序员若能深入理解TCP/IP协议则可大幅提高自身实力。

6.2 实现基于UDP的服务端、客户端

UDP服务器端、客户端不像TCP那样在连接状态下交换数据，因此与TCP不同，无需经过连接过程。

也就是不需要调用TCP连接过程中的 **listen** 函数 和 **accept** 函数

UDP服务器端和客户端均只需一个套接字，而TCP中服务器端需要N+1个套接字，其中N为相连客户端个数

6.2.1 基于UDP的数据I/O函数

1. 创建好TCP套接字后，传输数据时无需再添加地址信息。因为TCP套接字将保持与对方套接字的连接。因此， **TCP套接字总是知道目的地址信息**
2. UDP套接字不会保持连接状态，因此每次传递数据都要添加目标地址信息。

```
1 #include <sys/socket.h>
2
3 ssize_t sendto(int sock, void *buff, size_t nbytes, int
4 flags, struct sockaddr *to, socklen_t addrlen); //成功时
5 //返回传输的字节数，失败时返回-1
6 /*
7 1. sock: 用于传输数据的UDP套接字文件描述符
8 2. buff: 保存待传输数据的缓冲地址值
9 3. nbytes: 待传输的数据长度，以字节为单位
10 4. flags: 可选参数，若没有则传递0
11 5. to: 存有目标地址信息的sockaddr结构体变量的地址值
12 6. addrlen: 传递给参数to的地址值结构体变量长度
13 */
14
15 ssize_t recvfrom(int sock, void *buff, size_t nbytes,
16 int flags, struct sockaddr *from, socklen_t *addrlen);
17 //成功时返回接受的字节数，失败时返回-1
18 /*
19 1. sock: 用于接收数据的UDP套接字文件描述符
20 2. buff: 保存待接收数据的缓冲地址值
21 3. nbytes: 可接受的最大字节数，故无法超过参数buff所指的缓冲大小
22 4. flags: 可选参数，若没有则传递0
23 5. from: 存有发送端地址信息的sockaddr结构体变量的地址值
24 6. addrlen: 保存参数from的结构体变量长度的变量地址值
25 */
```

6.3 UDP客户端套接字的地址分配

在TCP客户端实现的时候，有一个通过connect同server相连接的过程。而在UDP客户端中则没有这一步骤。

UDP程序中，调用sendto函数传输数据前应完成对套接字的地址分配工作，因此调用bind函数。当然，bind函数在TCP程序中出现过，但bind函数不区分TCP和UDP，也就是说，在UDP程序中同样可以调用。另外，如果调用sendto函数时发现尚未分配地址信息，则在首次调用sendto函数时给相应套接字自动分配IP和端口。而且此时分配的地址一直保留到程序结束为止，因此也



① 可用来与其他UDP套接字进行数据交换。当然，IP用主机IP，端口号选尚未使用的任意端口号。

② 综上所述，调用sendto函数时自动分配IP和端口号，因此，UDP客户端中通常无需额外的地址分配过程。所以之前示例中省略了该过程，这也是普遍的实现方式。

6.4 UDP的数据传输特性和调用connect函数

TCP传输的数据不存在数据边界，而UDP数据传输中存在数据边界

UDP是具有数据边界的协议，传输中调用I/O函数的次数非常重要。因此，输入函数的调用次数应和输出函数的调用次数完全一致。这样才能保证接收全部已发送数据

例如：调用三次输出函数发送的数据必须通过调用三次输入函数才能接收完

证明必须在UDP通信过程中使I/O函数调用次数保持一致。



UDP 数据报 (Datagram)

UDP 套接字传输的数据包又称数据报，实际上数据报也属于数据包的一种。只是与 TCP 包不同，其本身可以成为 1 个完整数据。这与 UDP 的数据传输特性有关，UDP 中存在数据边界，1 个数据包即可成为 1 个完整数据，因此称为数据报。

6.4.1 已连接UDP套接字与未连接UDP套接字

TCP套接字中需注册待传输数据的目标IP和端口号，而UDP中则无需注册。因此，通过sendto函数传输数据的过程大致可分为以下3个阶段。

- 第1阶段：向UDP套接字注册目标IP和端口号。
- 第2阶段：传输数据。
- 第3阶段：删除UDP套接字中注册的目标地址信息。

每次调用sendto函数时重复上述过程。每次都变更目标地址，因此可以重复利用同一UDP套接字向不同目标传输数据。这种未注册目标地址信息的套接字称为未连接套接字，反之，注册了目标地址的套接字称为**连接connected套接字**。显然，UDP套接字默认属于未连接套接字。但UDP套接字在下述情况下显得不太合理：

“IP为211.210.147.82的主机82号端口共准备了3个数据，调用3次sendto函数进行传输。”

此时需重复3次上述三阶段。因此，要与同一主机进行长时间通信时，将UDP套接字变成已连接套接字会提高效率。上述三个阶段中，第一个和第三个阶段占整个通信过程近1/3的时间，缩短这部分时间将大大提高整体性能。

6.4.2 创建已连接UDP套接字

创建已连接UDP套接字的过程：只需要针对UDP套接字调用connect函数

```
connect(sock, (struct sockaddr*)&adr, sizeof(adr));
```

针对UDP套接字调用connect函数并不意味着要与对方UDP套接字连接，只是向UDP套接字注册**目标IP和端口信息**

对于已连接UDP套接字，不仅可以通过**sendto()**, **recvfrom()**还可以使用TCP套接字的**write**和**read**进行通信

```
/* sendto(sock, message, strlen(message), 0,
          (struct sockaddr*)&serv_addr, sizeof(serv_addr));
 */
write(sock, message, strlen(message));
```

注释部分：无连接的UDP套接字，sendto,recvfrom
有连接的套接字：write, read

第6章 基于 UDP 的服务器端/客户端

```
/*
adr_sz=sizeof(from_addr);
str_len=recvfrom(sock, message, BUF_SIZE, 0,
                  (struct sockaddr*)&from_addr, &adr_sz);
*/
str_len=read(sock, message, sizeof(message)-1);
```

6.5 问题探讨

1. UDP为什么比TCP快？为什么TCP数据传输可靠而UDP数据传输不可靠？

UDP比TCP更快的原因是它不存在的确认支持连续的包流。由于TCP连接总是确认一组数据包(无论连接是否完全可靠)，当数据包丢失时，每一次否定确认都必须进行重传

这也是TCP比UDP数据传输可靠的依据。面向连接，重传，流量控制等

2. (2) 下列不属于UDP特点的是？

- a. UDP不同于TCP，不存在连接的概念，所以不像TCP那样只能进行一对一的数据传输。
- b. 利用UDP传输数据时，如果有2个目标，则需要2个套接字。
- c. UDP套接字中无法使用已分配给TCP的同一端口号。X
- d. UDP套接字和TCP套接字可以共存。若需要，可以在同一主机进行TCP和UDP数据传输。
- e. 针对UDP函数也可以调用connect函数，此时UDP套接字跟TCP套接字相同，也需要经过3次握手过程。X

3. 何种情况下，UDP的性能优于TCP

网络带宽需求较小，而实时性要求高，大部分应用无需维持连接，需要低功耗

4. 客户端TCP套接字调用connect函数时自动分配IP和端口号，UDP不调用bind函数，而是在sendto的时候自动分配IP和端口号

5. UDP中调用connect函数的好处

- 节省在sendto()的时候，总是会自动分配IP和端口号的时间
- 对于经常进行数据传输的双方，UDP通过connect函数获取对方IP和端口号，可以通过write和read更加高效地完成数据传输工作

7. 优雅地断开套接字连接

在前面的示例中，我们总是调用close或closesocket函数单方面断开连接，同时断开两个I/O流

本章讨论的**优雅地断开连接方式**只断开其中一个流，而非同时断开两个流

7.1 基于TCP的半关闭

7.1.1 套接字和流

将建立套接字后可交换数据的状态看作一种流

一旦两台主机间建立了套接字连接，每个主机就会拥有独立的输入流和输出流

7.1.2 针对优雅断开的shutdown函数

```
1 #include <sys/socket.h>
2
3 int shutdown(int sock, int howto); //成功时返回0，失败时返回-1
4 //sock:需要断开的套接字文件描述符
5 //howto: 传递断开方式信息, SHUT_RD 断开输入流, SHUT_WR, 断开输出流, SHUT_RDWR
```

7.1.3 为什么要半关闭

存在的问题：对于数据传输的接收方，可能不知道要接收数据到何时。因此，只能无休止的调用输入函数，这有可能导致程序阻塞（调用的函数未返回）

解决办法：发送端在发送完数据以后传递一个EOF表示文件传输结束。

服务器如何传递EOF：

- 服务器端调用close函数可以同时关闭I/O流，但是有问题就是，这样可能无法再收到客户端想发送给服务器端的数据内容。
- 因此需要选择半关闭的 SHUT_WR

8. 域名及网络地址

8.1 常用的指令

1. ping 域名，得到某一个域名的ip
2. nslookup #查看默认的DNS服务器地址,linux下需要额外输入 server

使用域名的必要性：

因为IP地址可能会经常改变，因此不能将程序设计成要求用户手动输入可变动的IP地址和端口信息。

因此，可以通过将变动几率更小的域名作为程序参数，让程序根据域名自动获取IP地址。

需要IP和域名之间的转换函数

8.2 利用域名获取IP地址 gethostbyname

```
1 #include <netdb.h>
2
3 struct hostent *gethostbyname(const char* hostname); //成功是返回hostent结构体指针，失败时返回NULL指针
4
5 //struct hostent定义
6 struct hostent{
7     char* h_name; //official name
8     char** h_aliases; //alias list
9     int h_addrtype; //host address type
10    int h_length; //address length
11    char ** h_addr_list; //address list
12};
```

h_name

该变量中存有官方域名 (Official domain name)。官方域名代表某一主页，但实际上，一些著名公司的域名并未用官方域名注册。

h_aliases

可以通过多个域名访问同一主页。同一IP可以绑定多个域名，因此，除官方域名外还可指定其他域名。这些信息可以通过h_aliases获得。

h_addrtype

gethostbyname函数不仅支持IPv4，还支持IPv6。因此可以通过此变量获取保存在h_addr_list的IP地址的地址族信息。若是IPv4，则此变量存有AF_INET。

h_length

保存IP地址长度。若是IPv4地址，因为是4个字节，则保存4；IPv6时，因为是16个字节，故

保存16。

h_addr_list

这是最重要的成员。通过此变量以整数形式保存域名对应的IP地址。另外，用户较多的网站有可能分配多个IP给同一域名，利用多个服务器进行负载均衡。此时同样可以通过此变量获取IP地址信息。~~（X）~~

结构体成员中 h_addr_list 指向字符串指针数组（是由多个字符地址构成的数组），但字符串指针数组中的元素实际指向的是 in_addr(IPv4地址) 结构体变量地址值而非字符串。

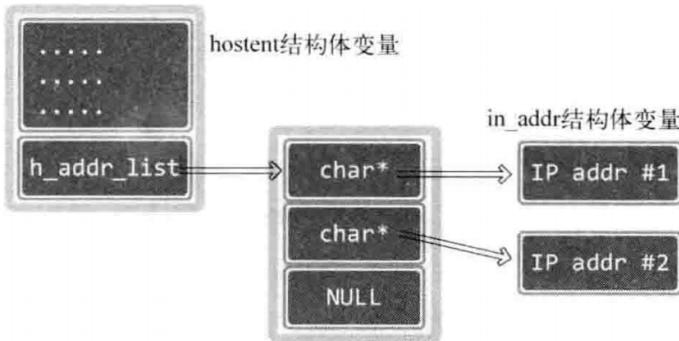


图8-3 h_addr_list结构体成员

Q1:为什么是char*而不是in_addr*: 因为hostent结构体并非只为IPv4准备，h_addr_list指向的数组中也可以保存IPv6地址信息

Q2:声明为void指针类型是否更合理? 从理论上来讲，当指针对象不明确时，更适合使用void指针类型，但是这个套接字相关函数是在void指针标准化之前定义的，所以只能采用char指针

因此需要进行类型转换

```

1 for(int i = 0; host->h_addr_list[i]; i++)
2   printf("IP addr %d : %s \n", i+1, inet_ntoa(*(struct
3     in_addr*)host->h_addr_list[i]));
  
```

```

[xiaotang@VM-0-5-centos src]$ vim gethostbyname.c
[xiaotang@VM-0-5-centos src]$ gcc gethostbyname.c -o hostname
[xiaotang@VM-0-5-centos src]$ ./hostname www.baidu.com
Official name: www.a.shifen.com
Aliases 1: www.baidu.com
Address type: AF_INET
IP addr 1: 182.61.200.7
IP addr 2: 182.61.200.6
  
```

8.3 利用IP地址获取域名gethostbyaddr

```

1 #include <netdb.h>
2
3 struct hostent* gethostbyaddr*(const char* addr,
4   socklen_t len, int family);
5 /* 
6 1. addr: 含有IP地址信息的in_addr结构体指针，为了同时传递IPv4地
7   址之外的其他信息，该变量的类型声明为char指针
8 2. len: 向第一个参数传递的地址信息的字节数，IPv4为4个字节，IPv6
9   为16个字节
10 3. family: 传递地址族信息，IPv4时为AF_INET，IPv6时为AF_INET6
11 */
  
```

9. 套接字的多种可选项

表9-1 可设置套接字的多种可选项

协议层	选项名	读	取	设	置
SOL_SOCKET	SO_SNDBUF	O		O	
	SO_RCVBUF	O		O	
	SO_REUSEADDR	O		O	
	SO_KEEPALIVE	O		O	
	SO_BROADCAST	O		O	
	SO_DONTROUTE	O		O	
	SO_OOBINLINE	O		O	
	SO_ERROR	O		X	
	SO_TYPE	O		X	
IPPROTO_IP	IP_TOS	O		O	
	IP_TTL	O		O	
	IP_MULTICAST_TTL	O		O	
	IP_MULTICAST_LOOP	O		O	
	IP_MULTICAST_IF	O		O	

9.1 套接字可选项和I/O缓冲大小 141

(续)

协议层	选项名	读	取	设	置
IPPROTO_TCP	TCP_KEEPALIVE	O		O	
	TCP_NODELAY	O		O	
	TCP_MAXSEG	O		O	

从表9-1中可以看出，套接字可选项是分层的。IPPROTO_IP层可选项是IP协议相关事项，IPPROTO_TCP层可选项是TCP协议相关的事项，SOL_SOCKET层是套接字相关的通用可选项。

9.1 getsockopt和setsockopt

```
1 #include <sys/socket.h>
2
3 int getsockopt(int sock, int level, int optname, void* optval, socklen_t* optlen); //成功时返回0, 失败时返回-1
4 /*
5  sock 用于查看选项套接字文件描述符
6  level 要查看的可选项的协议层
7  optname 要查看的可选项名
8  optval 保存查看结果的缓冲地址值
9  optlen 向第四个参数optval传递的缓冲大小, 调用函数后, 该变量中保
    存通过第四个参数返回的可选项信息的字节数
10 */
11
```

```

12 int setsockopt(int sock, int level, int optname, const
13 void* optval, socklen_t* optlen); //成功时返回0，失败时返
14 回-1
15 /*
16 sock 用于更改选项套接字文件描述符
17 level 要更改的可选项的协议层
18 optname 要更改的可选项名
19 optval 保存更改结果的缓冲地址值
20 optlen 向第四个参数optval传递的可选项信息的字节数 len =
21 sizeof(optval)
22 */

```

代码示例：

```

7. int main(int argc, char *argv[])
8. {
9.     int tcp_sock, udp_sock;
10.    int sock_type;
11.    socklen_t optlen;
12.    int state;
13.
14.    optlen=sizeof(sock_type);
15.    tcp_sock=socket(PF_INET, SOCK_STREAM, 0); 套接字创建
16.    udp_sock=socket(PF_INET, SOCK_DGRAM, 0);
17.    printf("SOCK_STREAM: %d \n", SOCK_STREAM);
18.    printf("SOCK_DGRAM: %d \n", SOCK_DGRAM);
19.
20.    state=getsockopt(tcp_sock, SOL_SOCKET, SO_TYPE, (void*)&sock_type, &optlen);
21.    if(state)   获取套接字相关信息
22.        error_handling("getsockopt() error!");
23.    printf("Socket type one: %d \n", sock_type);
24.
25.    state=getsockopt(udp_sock, SOL_SOCKET, SO_TYPE, (void*)&sock_type, &optlen);
26.    if(state)   获取套接字相关信息
27.        error_handling("getsockopt() error!");
28.    printf("Socket type two: %d \n", sock_type),
29.    return 0;
30. }
31

```

套接字类型的 `SO_TYPE` 是典型的只读可选项，即套接字类型只能在创建时设定，而不能通过 `setsockopt` 进行相关更改

9.2 SO_SNDBUF & SO_RCVBUF

如图5-2所示，调用write函数时，数据将移到输出缓冲，在适当的时候（不管是分别传送还是一次性传送）传向对方的输入缓冲。这时对方将调用read函数从输入缓冲读取数据。这些I/O缓冲特性可整理如下。

- I/O缓冲在每个TCP套接字中单独存在。
- I/O缓冲在创建套接字时自动生成。
- 即使关闭套接字也会继续传递输出缓冲中遗留的数据。
- 关闭套接字将丢失输入缓冲中的数据。

`SO_RCVBUF`是输入缓冲大小相关可选项，`SO_SNDBUF`是输出缓冲大小相关可选项。可以通过这两个选项 `读取或设定` 当前I/O缓冲大小

get_buf.c代码测试：

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <sys/socket.h>
5. void error_handling(char *message);
6.
7. int main(int argc, char *argv[])
8. {
9.     int sock;
10.    int snd_buf, rcv_buf, state;
11.    socklen_t len;
12.
13.    sock=socket(PF_INET, SOCK_STREAM, 0);
14.    len=sizeof(snd_buf);
15.    state=getsockopt(sock, SOL_SOCKET, SO_SNDBUF, (void*)&snd_buf, &len);

    读取sock的相关选项          关键字?

16.    if(state)
17.        error_handling("getsockopt() error");
18.
19.    len=sizeof(rcv_buf);
20.    state=getsockopt(sock, SOL_SOCKET, SO_RCVBUF, (void*)&rcv_buf, &len);
21.    if(state)
22.        error_handling("getsockopt() error");
23.
24.    printf("Input buffer size: %d \n", rcv_buf);
25.    printf("Output buffer size: %d \n", snd_buf);
26.    return 0;
27. }
28.
29. void error_handling(char *message)
30. {
31.     fputs(message, stderr);
32.     fputc('\n', stderr);
33.     exit(1);
34. }
```

setbuf.c代码测试

```
1. #include <"头声明与get_buf.c一致, 故省略。">
2. void error_handling(char *message);
3.
4. int main(int argc, char *argv[])
5. {
6.     int sock;
7.     int snd_buf=1024*3, rcv_buf=1024*3;
8.     int state; 定义setsockopt的读取状态
9.     socklen_t len;
10.    ipv4地址          TCP协议
11.    sock=socket(PF_INET, SOCK_STREAM, 0);
12.    state=setsockopt(sock, SOL_SOCKET, SO_RCVBUF, (void*)&rcv_buf, sizeof(rcv_buf));
13.    if(state)
14.        error_handling("setsockopt() error!");
15.
16.    state=setsockopt(sock, SOL_SOCKET, SO_SNDBUF, (void*)&snd_buf, sizeof(snd_buf));
17.    if(state)
18.        error_handling("setsockopt() error!");
```

```

19.
20.     len=sizeof(snd_buf);
21.     state=getsockopt(sock, SOL_SOCKET, SO_SNDBUF, (void*)&snd_buf, &len); set后get读取
22.     if(state)
23.         error_handling("getsockopt() error!");
24.
25.     len=sizeof(rcv_buf);
26.     state=getsockopt(sock, SOL_SOCKET, SO_RCVBUF, (void*)&rcv_buf, &len);
27.     if(state)
28.         error_handling("getsockopt() error!");
29.
30.     printf("Input buffer size: %d \n", rcv_buf);
31.     printf("Output buffer size: %d \n", snd_buf);
32.     return 0;
33. }
34.
35. void error_handling(char *message)
36. {
37.     fputs(message, stderr);
38.     fputc('\n', stderr);
39.     exit(1);
40. }

```

输出结果跟我们预想的完全不同，但也算合理。缓冲大小的设置需谨慎处理，因此不会完全按照我们的要求进行，只是通过调用setsockopt函数向系统传递我们的要求。如果把输出缓冲设置为0并如实反映这种设置，TCP协议将如何进行？如果要实现流控制和错误发生时的重传机制，至少要有一些缓冲空间吧？上述示例虽没有100%按照我们的请求设置缓冲大小，但也大致反映出了通过setsockopt函数设置的缓冲大小。

9.3 SO_REUSEADDR

9.3.1 发生地址分配错误

1. 当服务器端向客户端发送FIN消息的时候，那么服务器端在重新运行时将产生问题，如果用同一端口还哦重新运行服务器端，会输出**bind() error**消息，并且无法再次运行。但是经过几分钟后，即可重新运行服务器端
2. 通过客户端通知服务器端断开TCP连接，那么可以及时地关闭文件及套接字。重新运行服务器端也不会出现上述问题。

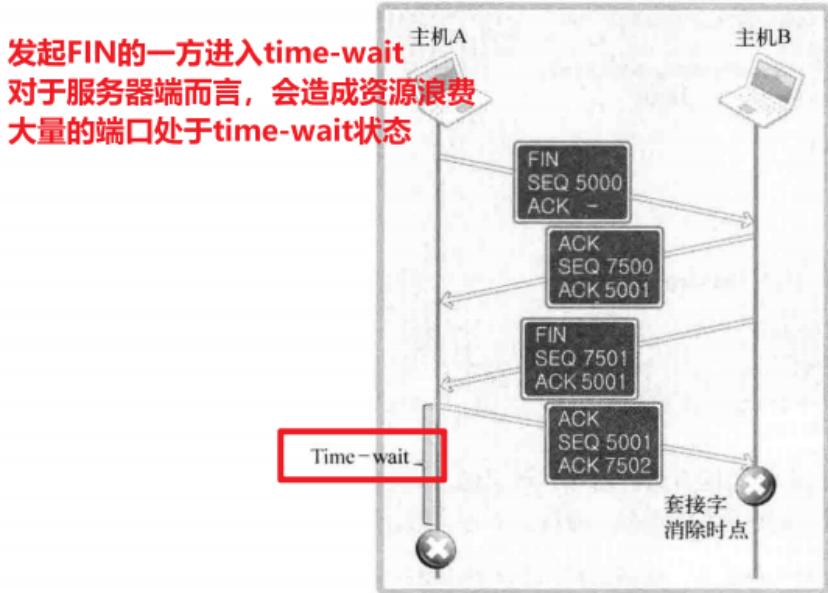


图9-1 Time-wait状态下的套接字

假设图9-1中主机A是服务器端，因为是主机A向B发送FIN消息，故可以想象成服务器端在控制台输入CTRL+C。但问题是，套接字经过四次握手过程后并非立即消除，而是要经过一段时间的Time-wait状态。当然，只有先断开连接的（先发送FIN消息的）主机才经过Time-wait状态。因此，若服务器端先断开连接，则无法立即重新运行。套接字处在Time-wait过程时，相应端口是正在使用状态。因此，就像之前验证过的，bind函数调用过程中当然会发生错误。

9.3.2 解决方案：地址再分配机制

- 假设服务器端发生故障从而紧急停止，那么需要快速重启服务器端。而处于time-wait的服务器需要等待time-wait
- 如果主动断开连接方最后的ACK没有传递成功，那么另一方会再次重传FIN消息，而收到FIN消息的主机将重启TIME-WAIT计时器。因此，可能会持续很久

解决方案：在套接字的可选项参数中更改 `SO_REUSEADDR` 的状态，适当调整参数，将time-wait状态下的套接字端口号重新分配给新的套接字。

`SO_REUSEADDR` 的默认值为0，表示无法分配time-wait状态下的套接字端口号，因此，需要将这个值更改为1

```
1 | setsockopt(serv_sock, SOL_SOCKET, SO_REUSEADDR,
  | (void*)&option, optlen); //参考TCP/IP网络编程158页
```

9.4 TCP_NODELAY

9.4.1 Nagle算法（应用于TCP层）

为了防止因数据包过多而发生网络过载，TCP套接字默认使用Nagle算法交换数据，最大限度地进行缓冲。直到收到ACK

不使用Nagle算法将对网络流量产生负面影响，即使只传输1个字节的数据，因为数据报文的头部会包括许多信息，可能有几十个字节。

因此，为了提高网络传输效率，必须使用Nagle算法

TCP套接字默认使用Nagle算法交换数据，因此最大限度地进行缓冲，直到收到ACK。图9-3左侧正是这种情况。为了发送字符串“Nagle”，将其传递到输出缓冲。这时头字符“N”之前没有其他数据（没有需接收的ACK），因此立即传输。之后开始等待字符“N”的ACK消息，等待过程中，剩下的“agle”填入输出缓冲。接下来，收到字符“N”的ACK消息后，将输出缓冲的

“agle”装入一个数据包发送。也就是说，共需传递4个数据包以传输1个字符串。

根据数据传输的特性，当网络流量未受太大影响时，不使用Nagle算法要比使用时传输速度快。

传输大文件数据：将文件数据传入输出缓冲不会花太多时间，因此即使不使用Nagle算法，也会在装满输出缓冲时传输数据包。这不仅不会增加数据包的数量，反而会在无须等待ACK的前提下连续传输。因此可以大大提高传输速度

一般情况下，不使用Nagle算法可以提高传输速度，但是无条件放弃使用Nagle算法，就会增加过多的网络流量，反而影响传输。因此，在未准确判断数据特行时不应禁用Nagle算法

9.4.2 禁用Nagle算法

禁用方法非常简单。从下列代码也可看出，只需将套接字可选项TCP_NODELAY改为1（真）即可。

```
int opt_val=1;
setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (void *) &opt_val, sizeof(opt_val));
```

可以通过TCP_NODELAY的值查看Nagle算法的设置状态。

```
int opt_val;
socklen_t opt_len;
```

```
opt_len=sizeof(opt_val);
getsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (void*) &opt_val, &opt_len);
```

如果正在使用Nagle算法，opt_val变量中会保存0；如果已禁用Nagle算法，则保存1。

10. 多进程服务器

10.1 并发服务器端的实现方法

即使有可能延长服务时间，也使其能够同时向所有发起请求的客户端提供服务

并且，网络程序中数据通信时间比cpu运算时间占比更大，因此，向多个客户端提供服务是一种有效利用CPU的方式

具有代表性的并发服务器实现模式与方法：

1. 多进程服务器：通过创建多个进程提供服务
2. 多路复用服务器：通过捆绑并统一管理I/O对象提供服务
3. 多线程服务器：通过生成与客户端等量的线程提供服务

10.2 多进程服务器

10.2.1 理解进程

进程：占用内存空间的正在运行的程序

从操作系统的角度来看，进程是程序流的基本单位，若创建多个进程，则操作系统将同时运行。

有时一个程序运行过程中也会产生多个进程

10.2.2 通过程序创建进程的方法

1. 通过 `ps` 命令查看进程
2. 通过 `fork` 函数创建进程

```
1 #include <unistd.h>
2
3 pid_t fork(void); //成功时返回进程ID，失败时返回-1，返回值类型pid_t
```

`fork` 函数将创建调用的进程副本，即并非根据完全不同的程序创建进程，而是复制正在运行的，调用 `fork` 函数的进程。因为通过同一个进程、复制相同的内存空间。之后的程序流就需要根据 `fork` 函数的返回值区分

并且，两个进程都将执行 `fork` 函数调用后的语句。通过 `fork` 函数返回值加以区分。

- 父进程（原进程，`fork` 函数的主体）：`fork` 函数返回子进程ID
- 子进程：`fork` 函数返回0

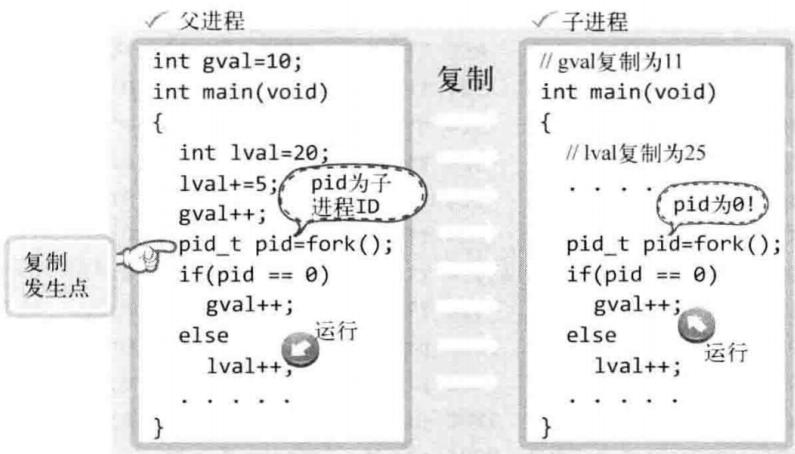


图10-1 fork函数的调用

10.3 进程和僵尸进程

进程完成工作后应被销毁，但有时候这些进程将变成僵尸进程，占用系统中的重要资源。这种状态下的进程称作“僵尸进程”

10.3.1 产生僵尸进程的原因

向exit函数传递的参数值和main函数的return语句返回的值都会传递给操作系统。而操作系统不会销毁子进程，直到把这些值传递给产生孩子进程的父进程。处在这种状态下的进程就是僵尸进程。**将子进程变成僵尸进程的是操作系统**

向exit函数传递的参数值和main函数的return语句返回的值都会传递给操作系统。而操作系统不会销毁子进程，直到把这些值传递给产生孩子进程的父进程。处在这种状态下的进程就是僵尸进程。也就是说，将子进程变成僵尸进程的正是操作系统。既然如此，此僵尸进程何时被销毁呢？其实已经给出提示。

这才是为什么后面可以通过waitpid, sigaction进行僵尸进程消除的原因。
通过及时向父进程传递子进程的返回值

10.3.2 销毁僵尸进程1：利用wait函数

为了销毁子进程，父进程应主动请求获取子进程的返回值

```

1 #include <sys/wait.h>
2
3 pid_t wait(int* statloc); // 成功时返回终止的子进程ID，失败时返回-1

```

调用此函数时如果已有子进程终止，那么子进程终止时传递的返回值（exit函数的参数值，main函数的return返回值）将保存到该函数的参数所指内存空间。但函数参数指向的单元中还包含其他信息。因此需要通过下列宏进行分离：

1. **WIFEXITED** 子进程正常终止时返回真

2. `WEXITSTATUS` 返回子进程的返回值

因此，在向wait函数传递变量status的地址时，调用wait函数后应该通过下列形式调用

```
1 if(WIFEXITED(status))
2 {
3     puts("Normal termination!");
4     printf("Child pass num: %d",
5            WEXITSTATUS(status));
6 }
```

如果没有已终止的子进程，那么程序将阻塞直到有子进程终止。因此考虑使用 `waitpid` 函数

10.3.3 通过 `waitpid` 函数销毁僵尸进程

wait函数会引起程序阻塞，因此可以考虑调用 `waitpid` 函数

```
1 #include <sys/wait.h>
2
3 pid_t waitpid(pid_t pid, int* statloc, int options); // 成功时返回终止的子进程ID或0，失败时返回-1
4 /*
5 pid: 等待终止的目标子进程的id，若传递-1，则与wait函数相同，可以等待任一子进程终止
6 statloc: 与wait函数的statloc参数具有相同含义
7 options: 传递头文件sys/wait.h中声明的常量WNOHANG，即使没有终止的子进程也不会进入阻塞状态，而是返回0并退出函数
8 */
```

10.4 信号处理

10.4.1 信号与函数

```
1 #include <signal.h>
2
3 void (*signal(int signo, void (*func)(int)))(int); //为了在产生信号时调用，返回之前注册的函数指针
4 //在发生第一个参数的特殊情况信息时，要调用第二个传入的参数
5 /*
6 1. void (*func)(int) 作为函数参数传入的函数指针，要求返回类型
7 void，参数类型int
8 2. 特殊情况信息
9 - SIGALRM: 已到通过调用alarm函数注册的时间
10 - SIGINT: 输入ctrl + c
11 - SIGCHLD: 子进程终止
12 3. signal函数名
13 4. 返回类型，参数为int型，，返回void型的函数指针
14 */
15
```

因此，对于上述定义的信号注册函数，若要实现子进程终止时调用mychild函数

那么signal函数调用如下

```
1 signal(SIGCHLD, mychild); //signal已经声明为函数指针
2 //函数指针调用参考
3 #include <stdio.h>
4
5 int max(int x, int y)
6 {
7     return x > y ? x : y;
8 }
9
10 int main(void)
11 {
12     /* p 是函数指针 */
13     int (* p)(int, int) = & max; // &可以省略
14     int a, b, c, d;
15
16     printf("请输入三个数字:");
17     scanf("%d %d %d", &a, &b, &c);
18
19     /* 与直接调用函数等价，d = max(max(a, b), c) */
20     d = p(p(a, b), c);
21 }
```

```
22 printf("最大的数字是: %d\n", d);  
23  
24 return 0;  
25 }
```

10.4.2 alarm函数

```
#include <unistd.h>  
  
unsigned int alarm(unsigned int seconds);  
→ 返回 0 或以秒为单位的距 SIGALRM 信号发生所剩时间。
```

如果调用该函数的同时向它传递一个正整型参数，相应时间后（以秒为单位）将产生SIGALRM信号。~~若向该函数传递0，则之前对SIGALRM信号的预约将取消。如果通过该函数预约信号后未指定该信号对应的处理函数，则（通过调用signal函数）终止进程，不做任何处理。~~
希望引起注意。**信号槽机制，并且默认终止调用alarm的进程**

```
5 void timeout(int sig)  
6 {  
7     if(sig==SIGALRM)  
8         puts("Time out!");  
9     alarm(2); ←  
10 }  
11 void keycontrol(int sig)  
12 {  
13     if(sig==SIGINT)  
14         puts("CTRL+C pressed");  
15 }  
16.  
17 int main(int argc, char *argv[]){  
18 {  
19     int i;  
20     signal(SIGALRM, timeout);  
21     signal(SIGINT, keycontrol);  
22     alarm(2);  
23.  
24     for(i=0; i<3; i++)  
25     {  
26         puts("wait...");  
27         sleep(100); //等待100秒，3次，在等待的过程中，signal函数信号检测也在执行  
28     }  
29     return 0;  
30 }
```

代码说明

- 第5、11行：分别定义信号处理函数。这种类型的函数称为信号处理器（Handler）。
- 第9行：~~为了每隔2秒重复产生SIGALRM信号，在信号处理器中调用alarm函数。~~
- 第20、21行：注册SIGALRM、SIGINT信号及相应处理器。
- 第22行：预约2秒后发生SIGALRM信号。
- 第27行：为了查看信号产生和信号处理器的执行并提供每次100秒、共3次的等待时间，在循环中调用sleep函数。也就是说，再过300秒、约5分钟后终止程序，这是相当长的一段时间，~~但实际执行时只需不到10秒。关于其原因稍后再解释。~~

猜测是因为在等待的过程中，alarm产生的SIGALRM信号执行

发生信号时，将唤醒由于调用sleep函数而进入阻塞状态的进程。

调用函数的主体是操作系统，但进程处于睡眠状态时无法调用函数。因此，产生信号时，为了调用信号处理器，将唤醒由于调用sleep函数而进入阻塞状态的进程。而且进程一旦被唤醒，就不会再进入睡眠状态。

10.4.3 利用sigaction函数进行信号处理

```

#include <signal.h>

int sigaction(int signo, const struct sigaction * act, struct sigaction * oldact);

→ 成功时返回 0，失败时返回-1。

```

- signo 与signal函数相同，传递信号信息。
- act 对应于第一个参数的信号处理函数（信号处理器）信息。
- oldact 通过此参数获取之前注册的信号处理函数指针，若不需要则传递0。

声明并初始化sigaction结构体变量以调用上述函数，该结构体定义如下。

```

struct sigaction
{
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}

```

此结构体的sa_handler成员保存信号处理函数的指针值（地址值）。sa_mask和sa_flags的所有位均初始化为0即可。这2个成员用于指定信号相关的选项和特性，而我们的目的主要是防止产生僵尸进程，故省略。理解这些参数所需参考书将在后面给出。

sigaction代码示例

```

1. #include <stdio.h>
2. #include <unistd.h>
3. #include <signal.h>
4.
5. void timeout(int sig) 信号处理函数
6. {
7.     if(sig==SIGALRM)
8.         puts("Time out!");
9.     alarm(2);
10. }
11.
12. int main(int argc, char *argv[])
13. {
14.     int i;
15.     struct sigaction act; struct sigaction结构体变量
16.     act.sa_handler=timeout;
17.     sigemptyset(&act.sa_mask); 变量属性设置
18.     act.sa_flags=0;

19.     sigaction(SIGALRM, &act, 0);
20.     ↑
21.     alarm(2);
22.
23.     for(i=0; i<3; i++)
24.     {
25.         puts("wait...");
26.         sleep(100);
27.     }
28.     return 0;
29. }

```

10.4.4 利用信号处理技术消灭僵尸进程

子进程终止时，将产生 SIGCHLD 信号，因此可以通过 sigaction 捕捉这一信号并执行响应的操作

10.5 实现并发服务器

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <signal.h>
6 #include <sys/wait.h>
7 #include <arpa/inet.h>
8 #include <sys/socket.h>
9
10#define BUF_SIZE 30
11void error_handling(char *message);
12void read_childproc(int sig);
13
14int main(int argc, char* argv[])
15{
16    int serv_sock, clnt_sock;
17    struct sockaddr_in serv_addr, clnt_addr;
18
19    pid_t pid;
20    struct sigaction act;
21    socklen_t adr_sz;
22    int str_len, state;
23    char buf[BUF_SIZE];
24    if(argc!=2) {
25        printf("Usage: %s <port>\n", argv[0]);
26        exit(1);
27    }
28    act.sa_handler = read_childproc;
29    sigemptyset(&act.sa_mask);
30    act.sa_flags = 0;
31    state = sigaction(SIGCHLD, &act, 0);
32    serv_sock = socket(PF_INET, SOCK_STREAM, 0);
33    memset(&serv_addr, 0, sizeof(serv_addr));
34    serv_addr.sin_family = AF_INET;
35    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
36    serv_addr.sin_port = htons(atoi(argv[1]));
37
38    if(bind(serv_sock, (struct sockaddr*)&serv_addr,
39            sizeof(serv_addr)) == -1)
```

```
39         error_handling("bind() error");
40     if(listen(serv_sock, 5) == -1)
41         error_handling("listen() error");
42
43     while(1)
44     {
45         adr_sz = sizeof(cInt_addr);
46         cInt_sock = accept(serv_sock, (struct
47             sockaddr*)&cInt_addr, &adr_sz);
48         if(cInt_sock == -1) continue; //当前没有客户端连接
49         else puts("new client connected...");
50
51         //创建子进程来处理新的客户端
52         pid = fork();
53         if(pid == -1)
54             { close(cInt_sock); continue; }
55         if(pid == 0)
56             {
57                 close(serv_sock); //关闭子进程中的服务端套接字
58                 while((str_len == read(cInt_sock, buf,
59                     BUF_SIZE)) !=0)
60                     write(cInt_sock, buf, str_len);
61
62                 close(cInt_sock);
63                 puts("client disconnected..."); //关闭父进程中的客户端套接字
64             }
65         close(serv_sock);
66         return 0;
67     }
68
69 void read_childproc(int sig)
70 {
71     pid_t pid;
72     int status;
73     pid = waitpid(-1, &status, WNOHANG);
74     printf("removed proc id: %d \n", pid);
75 }
76
```

```

77 | void error_handling(char* message)
78 {
79     fputs(message, stderr);
80     fputc('\n', stderr);
81     exit(1);
82 }

```

fork 函数复制文件描述符的时候，父进程会将所有套接字都复制给子进程。

因此会导致同一个端口对应多个套接字。而一个只有当这多个文件描述符都终止或销毁的时候，套接字才能够被销毁。进而端口关闭。因此，调用**fork**函数后，要将无关的套接字文件描述符都关闭。

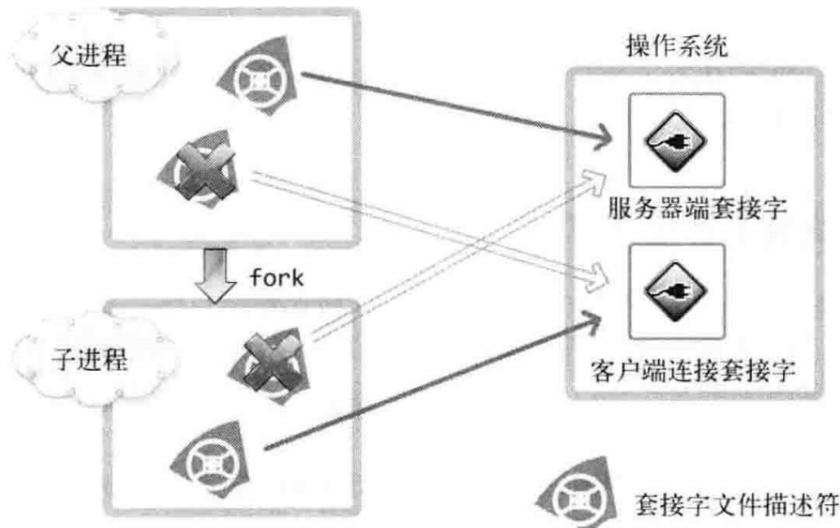


图10-4 整理复制的文件描述符

10.5.1 分割TCP的I/O程序

优点：通过创建多个进程，分割数据收发流程。另一个好处：可以提高频繁交换数据的程序性能

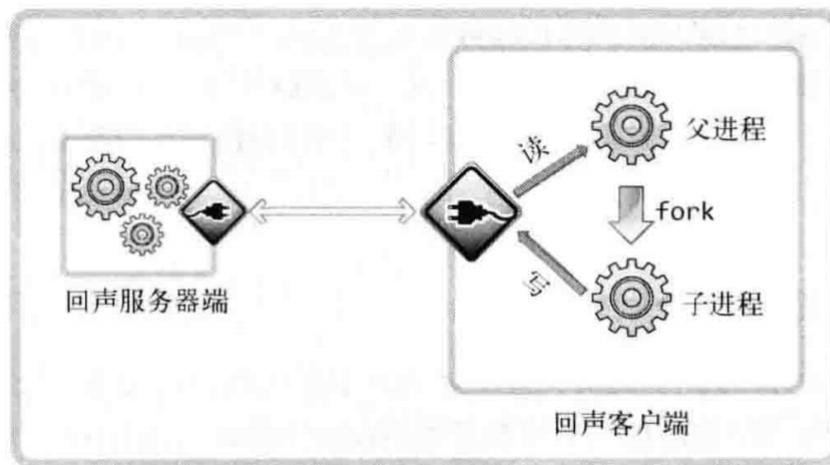


图10-5 回声客户端I/O分割模型

10.6 问题探讨



(1) 下列关于进程的说法错误的是?

- a. 从操作系统的角度上说，进程是程序运行的单位。
- b. 进程根据创建方式建立父子关系。
- c. 进程可以包含其他进程，即一个进程的内存空间可以包含其他进程。
- d. 子进程可以创建其他子进程，而创建出来的子进程还可以创建其子进程，但所有这些进程只与一个父进程建立父子关系。

(2) 调用fork函数将创建子进程，以下关于子进程的描述错误的是?

- a. 父进程销毁时也会同时销毁子进程。
- b. 子进程是复制父进程所有资源创建出的进程。
- c. 父子进程共享全局变量。
- d. 通过fork函数创建的子进程将执行从开始到fork函数调用为止的代码。

请说明进程变为僵尸进程的过程及预防措施：

在通过 `fork` 函数成功创建子进程以后，如果子进程任务执行完毕。那么会返回值给操作系统。直到返回值被其父进程接收为止，该（子）进程会一直作为僵尸进程存在。所以，为了防止这种情况的发生，父进程必须明确接收子进程结束时的返回值。

预防措施：

- `wait` 和 `waitpid`
- `sigaction` 函数

(5) 如果在未注册SIGINT信号的情况下输入Ctrl+C，将由操作系统默认的事件处理器终止程序。但如果直接注册Ctrl+C信号的处理器，则程序不会终止，而是调用程序员指定的事件处理器。编写注册处理函数的程序，完成如下功能：

“输入Ctrl+C时，询问是否确定退出程序，输入Y则终止程序。”

另外，编写程序使其每隔1秒输出简单字符串，并适用于上述时间处理器注册代码。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4
5 void ctrl_handler(int sig);
```

```

6
7 int main(int argc, char *argv[])
8 {
9     struct sigaction act;
10    act.sa_handler=ctrl_handler;
11    sigemptyset(&act.sa_mask);
12    act.sa_flags=0;
13    sigaction(SIGINT, &act, 0);
14
15    while(1)
16    {
17        sleep(1);
18        puts("Have a nice day~");
19    }
20
21    return 0;
22 }
23
24
25 void ctrl_handler(int sig)
26 {
27     char ex;
28     fputs("Do you want exit(Y to exit)? ", stdout);
29     scanf("%c", &ex);
30     if(ex=='y' || ex=='Y')
31         exit(1);
32 }
```

11. 进程间通信

进程间通信指两个不同进程间可以交换数据，因此操作系统中应该提供两个进程可以同时访问的内存空间

进程具有完全独立的内存结构，就连通过fork函数创建的子进程也不会与父进程共享内存空间。因此进程间通信只能通过其他特殊方法完成

11.1 进程间通信方式1：通过管道实现进程间通信

两个进程通过操作系统提供的内存空间进行通信

```

1 #include <unistd.h>
2
```

```

3 int pipe(int filedes[2]); //成功时返回0, 失败时返回-1
4 //filedes[0]通过管道接收数据时使用的文件描述符, 即管道出口
5 //filedes[1]通过管道传输数据时使用的文件描述符, 即管道入口
6
7 //pipe1.c
8 #include <stdio.h>
9 #include <unistd.h>
10#define BUF_SIZE 30
11
12int main(int argc, char* argv[])
13{
14    int fds[2];
15    char str[] = "who are you?";
16    char buf[BUF_SIZE];
17    pid_t pid;
18
19    pipe(fds); //调用pipe函数创建管道, fds数组中保存用于I/O的
20    //文件描述符
21    pid = fork();
22    if(pid == 0)
23    {
24        write(fd[1], str, sizeof(str)); //管道入口, 发送数
25        //据
26    }
27    else
28    {
29        read(fds[0], buf, BUF_SIZE); //管道出口, 接收数据
30        puts(buf);
31    }
32    return 0;
33}

```

11.1.1 通过管道进行进程间双向通信

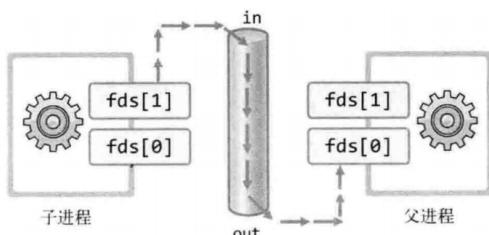


图11-2 示例pipe1.c的通信路径 **单向通信模型**

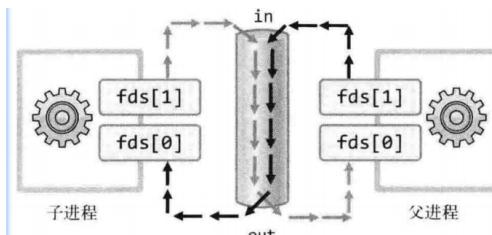


图11-3 双向通信模型1

```
1 #include <stdio.h>
```

```

2 #include <unistd.h>
3 #define BUF_SIZE 30
4
5 int main(int argc, char* argv[])
6 {
7     int fds[2];
8     char str1[] = "who are you?";
9     char str2[] = "Thank you for your message";
10    char buf[BUF_SIZE];
11    pid_t pid;
12
13    pipe(fds); //调用pipe函数创建管道, fds数组中保存用于I/O的
14    //文件描述符
15    pid = fork();
16    if(pid == 0)
17    {
18        write(fds[1], str1, sizeof(str1)); //管道入口, 发
19        //送数据
20        sleep(2); //注释掉之后会发生错误, 因为数据进入管道之后
21        //会成为无主数据, 也就是任何进程都可以将管道中的数据read出来。即子进
22        //程write之后也可以立即read自己所写数据
23        read(fd[0], buf, BUF_SIZE);
24        printf("Child proc output: %s \n", buf);
25    }
26    else
27    {
28        read(fds[0], buf, BUF_SIZE); //管道出口, 接收数据
29        printf("Parent proc output: %s \n", buf);
30        write(fds[1], str2, sizeof(str2)); //管道入口, 发
31        //送数据
32        sleep(3);
33    }
34    return 0;
35 }
```

通过一个管道进行双向通信会出现上述存在的问题。因此通常**创建两个管道来实现双向通信**

双管道实现双向通信代码示例

```

1 #include <stdio.h>
2 #include <unistd.h>
```

```

3 #define BUF_SIZE 30
4
5 int main(int argc, char* argv[])
6 {
7     int fds1[2], fds2[2];
8     char str1[] = "who are you?";
9     char str2[] = "I am the parent proc!";
10    char buf[BUF_SIZE];
11    pid_t pid;
12
13    pipe(fds1); //调用pipe函数创建管道, fds数组中保存用于I/O
的文件描述符
14    pipe(fds2);
15    pid = fork();
16    if(pid == 0)
17    {
18        write(fds1[1], str1, sizeof(str1)); //管道入口,
发送数据
19        read(fds2[0], buf, BUF_SIZE);
20        printf("Child proc output: %s \n", buf);
21    }
22    else
23    {
24        read(fds1[0], buf, BUF_SIZE); //管道出口, 接收数
据
25        printf("Parent proc output: %s \n", buf);
26        write(fds2[1], str2, sizeof(str2)); //管道入口,
发送数据
27    }
28    return 0;
29 }

```

11.2 问题探讨

- 什么是进程间通信？分别从概念上和内存的角度进行说明

进程间通信指的是两个不同进程间可以交换数据，

为了完成这一点，操作系统中应提供两个进程可以同时访问的内存空间

- 进程间通信需要特殊的IPC机制，这是由操作系统提供的。进程间通信时，为何需要操作系统的帮助

要想实现IPC机制，需要共享的内存，但由于两个进程之间不共享内存，因此需要操作系统的帮助，也就是说，两进程共享的内存空间必须由操作系统来提供

3. 为了完成进程间通信，2个进程需同时连接管道。那2个进程如何连接到同一管道？

pipe函数通过输入参数返回管道的输入输出文件描述符。这个文件描述符在fork函数中复制到了其子进程，因此，父进程和子进程可以同时访问同一管道。

4. 管道允许进行2个进程间的双向通信。双向通信中需要注意哪些内容？

管道并不管理进程间的数据通信。因此，如果数据流入管道，任何进程都可以读取数据。因此，**要合理安排共享空间的输入和读取**

更好的方法是通过两个管道实现双向通信

12. I/O复用

12.1 基于I/O复用的服务器端

多进程服务器端的缺点和解决办法

因为创建进程需要产生独立的内存空间，并且进程间也无法直接通信，只能通过管道通信。

复用技术的优点

1. 减少连线长度
2. 减少纸杯个数

复用技术分类：

1. 时分复用技术
2. 频分复用技术

+ 复用技术在服务器端的应用

纸杯电话系统引入复用技术后，可以减少纸杯数和连线长度。同样，服务器端引入复用技术可以减少所需进程数。为便于比较，先给出第10章的多进程服务器端模型，如图12-3所示。

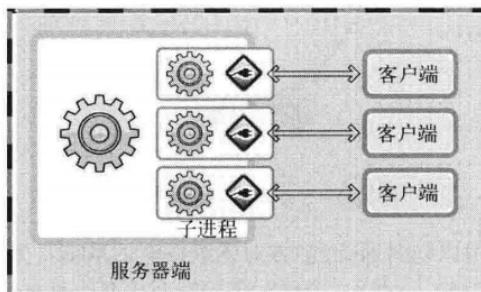


图12-3 多进程服务器端模型

图12-3的模型中引入复用技术，可以减少进程数。重要的是，无论连接多少客户端，提供服务的进程只有1个。

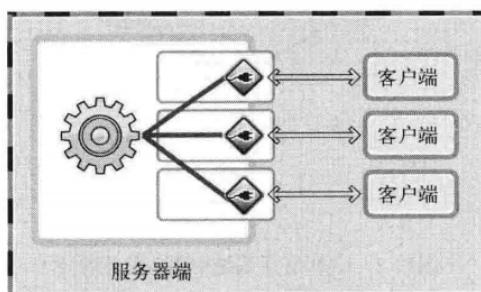


图12-4 I/O复用服务器端模型

12.2 理解select函数并实现服务器端

select函数的功能和调用顺序

使用select函数时可以将多个文件描述符集中到一起统一监视，项目如下。

- 是否存在套接字接收数据？
- 无需阻塞传输数据的套接字有哪些？
- 哪些套接字发生了异常？

提 示

监视项称为“事件”(event)

上述监视项称为“事件”。发生监视项对应情况时，称“发生了事件”。这是最常见的表达，希望各位熟悉。另外，本章不会使用术语“事件”，而与本章密切相关的第17章将使用该术语，希望大家理解“事件”的含义，以及“发生事件”的意义。

select函数调用过程：

1. 设置文件描述符
2. 指定监视范围
3. 设置超时
4. 调用select函数
5. 查看调用结果

12.2.1 设置文件描述符

使用 `fd_set` 可以将要监视的文件描述符集中到一起，并按照监视项（接收、传输、异常）分为三类

在 `fd_set` 变量中注册或更改值的操作都是通过宏完成的

- `FD_ZERO(fd_set * fdset)`：将 `fd_set` 变量的所有位初始化为 0。
- `FD_SET(int fd, fd_set * fdset)`：在参数 `fdset` 指向的变量中注册文件描述符 `fd` 的信息。
- `FD_CLR(int fd, fd_set * fdset)`：从参数 `fdset` 指向的变量中清除文件描述符 `fd` 的信息。
- `FD_ISSET(int fd, fd_set * fdset)`：若参数 `fdset` 指向的变量中包含文件描述符 `fd` 的信息，则返回“真”。

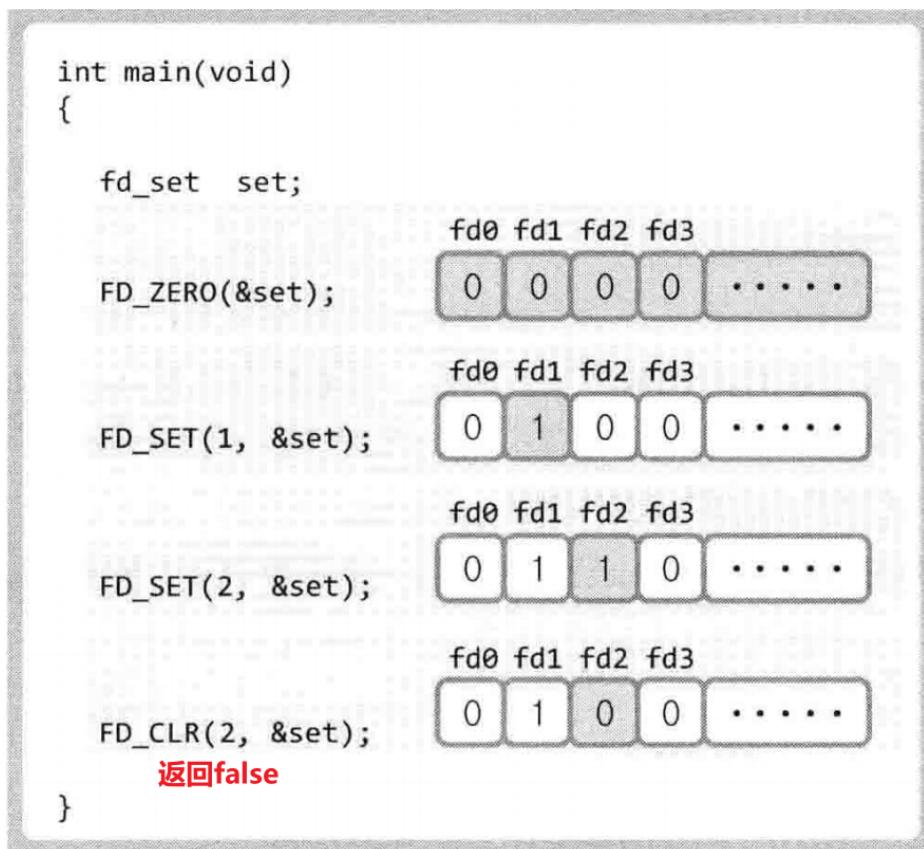


图12-7 `fd_set`相关函数的功能

12.2.2 设置监视范围及超时

`select` 函数只有在监视的文件描述符发生变化时才返回，如果未发生变化，就会进入阻塞状态。

指定超时事件可以避免无限阻塞状态

```
1 #include <sys/select.h>
2 #include <sys/time.h>
3
```

```

4 int select(int maxfd, fd_set* readset, fd_set*
5 writeset, fd_set* exceptset, const struct timeval*
6 timeout); //成功时返回大于0的值, 失败时返回-1
7 /*
8  maxfd: 监视对象文件描述符的数量 (文件描述符的监视范围)
9  readset: 将所有关注“是否存在待读取数据”的文件描述符注册到fd_set
10 型变量, 并传递其地址值
11  writeset: 将所有关注“是否可传输无阻塞数据”的文件描述符注册到
12 fd_set型变量, 并传递其地址值
13  exceptset: 将所有关注“是否发生异常”的文件描述符注册到fd_set型
14 变量, 并传递其地址值
15  timeout: 调用select函数后, 为防止陷入无限阻塞的状态, 传递超时
16 (time-out) 的信息。如果不设置超时, 则传递NULL参数
17  返回值: 发生错误返回-1, 超时返回0, 因发生关注的事件返回时, 返回大
18 于0的值, 该值表示发生事件的文件描述符个数
19 */
20
21
22 struct timeval
23 {
24     long tv_sec; //seconds
25     long tv_usec; //microseconds
26 }

```

文件描述符的变化:

是指监视的文件描述符中发生了相应的监视事件, 例如通过 `select` 的第二个参数传递的集合中存在需要读数据的描述符时, 就意味着文件描述符发生了变化

`select` 函数返回正整数时, 怎样获知哪些文件描述符发生了变化? 向 `select` 函数的第二到第四个参数传递的 `fd_set` 变量中将产生如图12-8所示变化, 获知过程并不难。

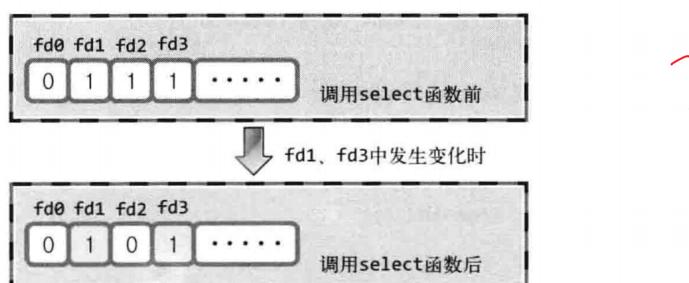


图12-8 fd_set变量的变化

由图12-8可知, `select` 函数调用完成后, 向其传递的 `fd_set` 变量中将发生变化。原来为1的所有位均变为0, 但发生变化的文件描述符对应位除外。因此, 可以认为值仍为1的位置上的文件描述符发生了变化。

//select函数代码示例

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/time.h>
4 #include <sys/select.h>
5
6 #define BUF_SIZE 30
7
8 int main(int argc, char* argv[])
9 {
10     fd_set reads, temps; //初始化fd_set变量
11     int result, str_len;
12     char buf[BUF_SIZE];
13
14     struct timeval timeout;
15
16     FD_ZERO(&reads);
17     FD_SET(0, &reads); // 0 is standard input
18
19     //timeout.tv_sec = 5;
20     //timeout.tv_usec = 5000;
21     //不能在此处设置超时，因为调用select函数后，结构体timeval的
22     成员的值会被替换为超时前剩余时间。
23     while(1)
24     {
25         temps = reads; //将准备好的fd_set变量的内容复制到
26         temps变量
27         //因为调用select函数后，除发生变化的文件描述符对应位
28         外，剩下的所有位都将初始化为0
29         //因此为了记住初始值，必须经过这种复制过程。这也是
30         select函数的通用方法
31         timeout.tv_sec = 5;
32         timeout.tv_usec = 0;
33         result = select(1, &temps, 0, 0, &timeout);
34         if(result == -1)
35             {puts("select() error"); break;}
36         else if(result == 0) puts("time out!");
37         else
```

```
38         str_len = read(0, buf, BUF_SIZE);
39         buf[str_len] = 0;
40         printf("message from console: %s",
41                buf);
42     }
43 }
44 return 0;
45 }
```

12.3 基于I/O复用的回声服务端

代码示例

```
1 #include <stdio.h>
2 #include <sys/socket.h>
3 #include <unistd.h>
4 #include <arpa/inet.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <sys/time.h>
8 #include <sys/select.h>
9
10#define BUF_SIZE 100
11void error_handling(char* buf);
12
13int main(int argc, char* argv[])
14{
15    int serv_sock, clnt_sock; //服务器端和客户端套接字
16    struct sockaddr_in serv_addr, clnt_addr; //套接字地址
17    struct timeval timeout;
18    fd_set reads, cpy_reads; //cpy_reads应该始终定义来保存
    初始状态
19
20    socklen_t adr_sz;
21    int fd_max, str_len, fd_num, i;
22
23    char buf[BUF_SIZE];
24    if(argc != 2){
25        printf("Usage: %s <port> \n", argv[0]);
26        exit(1);
```

```
27     }
28     serv_sock = socket(PF_INET, SOCK_STREAM, 0); //服务
29     memset(&serv_addr, 0, sizeof(serv_addr));
30     serv_addr.sin_family = AF_INET;
31     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
32     serv_addr.sin_port = htons(atoi(argv[1]));
33
34     if(bind(serv_sock, (struct sockaddr*)& serv_addr,
35             sizeof(serv_addr)) == -1)
36         error_handling("Bind() error");
37
38     if(listen(serv_sock, 5) == -1)
39         error_handling("Listen() error");
40     //设置初始监控范围
41     FD_ZERO(&reads);
42     FD_SET(serv_sock, &reads); //监听serv_sock的状态
43     fd_max = serv_sock;
44
45     while(1)
46     {
47         cpy_reads = reads;
48         timeout.tv_sec = 5;
49         timeout.tv_usec = 5000;
50
51         if((fd_num = select(fd_max+1, &cpy_reads, 0, 0,
52             &timeout)) == -1)
53             break;
54         if(fd_num == 0) continue; //监听超时
55         for(i = 0; i < fd_max + 1; i++)
56         {
57             if(FD_ISSET(i, &cpy_reads)) //读取对应文件描述
58                符的状态
59
60                 {
61                     if(i == serv_sock) //connection request
62                     {
63                         adr_sz = sizeof(cInt_addr);
64                         cInt_sock = accept(serv_sock,
65                             (struct sockaddr*)&cInt_addr, &adr_sz);
66                         FD_SET(cInt_sock, &reads);
67
68                     }
69                 }
70             }
71         }
72     }
73 }
```

```
62             if(fd_max < cInt_sock) fd_max =
63                 cInt_sock; //这里保证fd_set不会冲突
64             printf("connected client: %d \n",
65                 cInt_sock);
66         }
67     }
68     else
69     {
70         str_len = read(i, buf, BUF_SIZE);
71         if(str_len == 0) //close request
72         {
73             FD_CLR(i, &reads);
74             close(i);
75             printf("closed client: %d \n",
76                 i);
77         }
78     }
79 }
80 }
81 }
82 close(serv_sock);
83 return 0;
84 }
85
86 void error_handling(char *buf)
87 {
88     fputs(buf, stderr);
89     fputc('\n', stderr);
90     exit(1);
91 }
```

- 第40行：向要传到select函数第二个参数的fd_set变量reads注册服务器端套接字。这样，接收数据情况的监视对象就包含了服务器端套接字。客户端的连接请求同样通过传输数据完成。因此，服务器端套接字中有接收的数据，就意味着有新的连接请求。
- 第49行：在while无限循环中调用select函数。select函数的第三和第四个参数为空。只需根据监视目的传递必要的参数。
- 第54、56行：select函数返回大于等于1的值时执行的循环。第56行调用FD_ISSET函数，查找发生状态变化的（有接收数据的套接字的）文件描述符。
- 第58、63行：发生状态变化时，首先验证服务器端套接字中是否有变化。如果是服务器端套接字的变化，将受理连接请求。特别需要注意的是，第63行在fd_set变量reads中注册了与客户端连接的套接字文件描述符。FD_SET(clnt_sock, &reads);
- 第68行：发生变化的套接字并非服务器端套接字时，即有要接受的数据时执行else语句。但此时需要确认接收的数据是字符串还是代表断开连接的EOF。str_len == 0
- 第73、74行：接收的数据为EOF时需要关闭套接字，并从reads中删除相应信息。
- 第79行：接收的数据为字符串时，执行回声服务。FD_CLR(i, &reads);

12.4 问题探究

1. 关于fd_set

fd_set数组变量，用来表示文件描述符的监视状态。如果对应位为1，那么则表示该文件描述符时监视对象

2. 请解释复用技术的通用含义，并说明何为I/O复用

复用技术指为了提高物理设备的效率，用最少的物理要素传递最多数据时使用的技术。同样，I/O复用是指将需要I/O的套接字捆绑在一起，利用最少限度的资源来收发数据的技术

3. 多进程并发服务器的缺点有哪些？如何在I/O复用服务器端中弥补？

多进程并发服务器的服务方式是，每当客户端提出连接要求时，就会追加生成进程。但构建进程是一项非常有负担的工作，因此，向众多客户端提供服务存在一定的局限性。并且进程间通信，也比较困难。需要通过管道等方式而复用服务器则是将套接字的文件描述符捆绑在一起管理的方式，因此可以一个进程管理所有的I/O操作

4. select函数的观察对象中应包含服务器端套接字（监听套接字），那么应将其包含到哪一类监听对象集合？

服务器套接字的作用是监听有无连接请求，即判断接收的连接请求是否存在？应该将其包含到“读”类监听对象的集合中

13. 多种I/O函数

1. read()和write()
2. send()与recv()
3. readv()和writev()

13.1 linux中的send和recv函数

```
1 #include <sys/socket.h>
2 //和write相比，多了个可选项信息
3 ssize_t send(int sockfd, const void* buf, size_t
4 nbytes, int flags); //成功时返回发送的字节数，失败时返回-1
5 /*
6 sockfd: 表示与数据传输对象相连接的套接字文件描述符
7 buf: 保存待传输数据的缓冲地址值
8 nbytes: 待传输的字节数
9 flags: 传输数据时指定的可选项信息
10 */
11 ssize_t recv(int sockfd, void *buf, size_t nbytes, int
12 flags); //成功时返回接收到的字节数（收到EOF返回0）失败时返回-1
13 //示例
14 str_len = recv(recv_sock, buf, sizeof(buf) - 1, 0); //因为C风格字符串末尾有个'\0'
15 buf[str_len] = 0; //又添上末尾的0
```

表13-1 send&recv函数的可选项及含义

可选项 (Option)	含 义	send	recv
MSG_OOB	用于传输带外数据 (Out-of-band data)	•	•
MSG_PEEK	验证输入缓冲中是否存在接收的数据		•
MSG_DONTROUTE	数据传输过程中不参照路由 (Routing) 表，在本地 (Local) 网络中寻找目的地	•	
MSG_DONTWAIT	调用I/O函数时不阻塞，用于使用非阻塞 (Non-blocking) I/O	•	•
MSG_WAITALL	防止函数返回，直到接收全部请求的字节数		•

13.1.1 MSG_OOB：发送紧急消息

MSG_OOB 可选项可用于创建特殊发送方法和通道以发送紧急消息。

```
1 write(sock, "123", strlen("123"));
2 send(sock, "4", strlen("4"), MSG_OOB); //调用send函数时指
3 定MSG_OOB可选项即可
4 //MSG_OOB的消息接收：当收到MSG_OOB紧急消息时，操作系统将产生
5 SIGURG信号
6 void urg_handler(int signo)
7 {
```

```

8   char buf[BUF_SIZE];
9     str_len = recv(recv_sock, buf, sizeof(buf)-1,
10    MSG_OOB);
11    buf[str_len] = 0;
12    printf("Urgent Message: %s \n", buf);
13
14 struct sigaction act; //注册紧急信号处理函数
15 act.sa_handler = urg_handler;
16 sigemptyset(&act.sa_mask);
17 act.sa_flags = 0;
18
19 state = sigaction(SIGURG, &act, 0); //检测是否有MSG_OOB

```

上述示例中插入了未曾讲解的函数调用语句，关于此函数只讲解必要部分，过多解释将脱离本章主题（第17章将再次说明）。

`fcntl(recv_sock, F_SETOWN, getpid());`

`fcntl`函数用于控制文件描述符，但上述调用语句的含义如下：

“将文件描述符`recv_sock`指向的套接字拥有者（`F_SETOWN`）改为把`getpid`函数返回值用作ID的进程。”

各位或许感觉“套接字拥有者”的概念有些生疏。操作系统实际创建并管理套接字，所以从严格意义上说，“套接字拥有者”是操作系统。只是此处所谓的“拥有者”是指负责套接字所有事务的主体。上述描述可简要概括如下：

“文件描述符`recv_sock`指向的套接字引发的`SIGURG`信号处理进程变为将`getpid`函数返回值用作ID的进程。”

13

处理`SIGURG`信号时必须指定处理信号的进程，通过`getpid`函数返回调用次函数的进程ID

的确！令人遗憾的是，通过`MSG_OOB`可选项传递数据时不会加快数据传输速度，而且通过信号处理函数`urg_handler`读取数据时也只能读1个字节。剩余数据只能通过未设置`MSG_OOB`可选项的普通输入函数读取。这是因为TCP不存在真正意义上的“带外数据”。实际上，`MSG_OOB`中的`OOB`是指`Out-of-band`，而“带外数据”的含义是：

“通过完全不同的通信路径传输的数据。” send的时候添加`MSG_OOB`只能传输一个字节

即真正意义上的`Out-of-band`需要通过单独的通信路径高速传输数据，但TCP不另外提供，只利用TCP的紧急模式（Urgent mode）进行传输。

13.1.2 TCP紧急模式

`MSG_OOB`的真正意义在于督促数据接受对象尽快处理数据。但TCP的“保持传输顺序”的特性依然成立

指定`MSG_OOB`选项的数据包本身就是紧急数据包，并通过紧急指针表示紧急消息所在位置

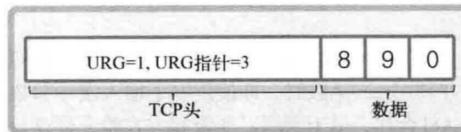


图13-2 设置URG的数据包

TCP数据包实际包含更多信息，但图13-2只标注了与我们的主题相关的内容。TCP头中含有如下两种信息。

- URG=1：载有紧急消息的数据包
- URG指针：紧急指针位于偏移量为3的位置

指定MSG_OOB选项的数据包本身就是紧急数据包，并通过紧急指针表示紧急消息所在位置。但通过图13-2无法得知以下事实：

“紧急消息是字符串890，还是90？如若不是，是否为单个字符0？”

但这并不重要。如前所述，除紧急指针的前面1个字节外，数据接收方将通过调用常用输入函数读取剩余部分。换言之，紧急消息的意义在于督促消息处理，而非紧急传输形式受限的消息。

13.1.3 检查输入缓冲

通过同时设置 `MSG_PEEK` 和 `MSG_DONTWAIT` 选项，来验证输入缓冲中是否存在接收的数据

- 设置了`MSG_PEEK`的`recv`函数，会读取输入缓冲数据，但不会删除输入缓冲数据

`PEEK`收发验证代码示例：

```

while(1)
{
    str_len=recv(recv_sock, buf, sizeof(buf)-1, MSG_PEEK|MSG_DONTWAIT);
    if(str_len>0)
        break;
}
    
```

通过非阻塞的形式来判断输入缓冲中是否存在数据
成功检测到数据，break

```

buf[str_len]=0;
printf("Buffering %d bytes: %s \n", str_len, buf);

str_len=recv(recv_sock, buf, sizeof(buf)-1, 0); //从输入缓冲中取走数据
buf[str_len]=0;
printf("Read again: %s \n", buf);
close(acpt_sock);
close(recv_sock);
return 0;
}

```

13.2 `readv`和`writev`函数

```

1 //可以通过readv和writev函数调用来减少I/O函数的调用次数
2 #include <sys/uio.h>
3 struct iovec
4 {
5     void* iov_base; //缓冲地址
6     size_t iov_len; //缓冲大小
7 }
8

```

```

9 ssize_t writev(int filedes, const struct iovec* iov,
10   int iovcnt); //成功时返回发送的字节数, 失败时返回-1
11 /*
12 filedes: 表示数据传输对象的套接字文件描述符。也可以像read函数一样
13 向其传递文件或标准输出描述符
14 iov: iovec结构体数组的地址, 包含待发送数据的位置和大小信息
15 iovcnt: 向第二参数传递的数组长度
16 */
17
18 ssize_t readv(int filedes, const struct iovec* iov, int
19   iovcnt); //成功时返回接受的字节数, 失败时返回-1

```

文件描述符:

- 1: 向控制台输出数据
- 0: 获取控制台输入数据

可以看到, 结构体iovec由保存待发送数据的缓冲 (char型数组) 地址值和实际发送的数据长度信息构成。给出上述函数的调用示例前, 先通过图13-4了解该函数的使用方法。

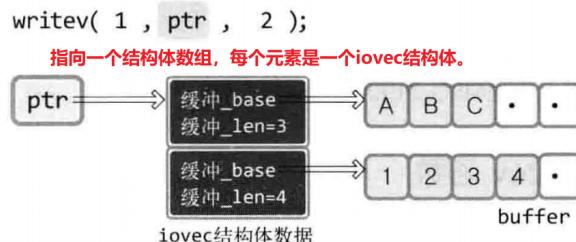


图13-4 write & iovec

图13-4中writev的第一个参数1是文件描述符, 因此向控制台输出数据。ptr是存有待发送数据信息的iovec数组指针。第三个参数为2, 因此, 从ptr指向的地址开始, 共浏览2个iovec结构体变量, 发送这些指针指向的缓冲数据。接下来仔细观察图中的iovec结构体数组。ptr[0] (数组第一个元素) 的iov_base指向以A开头的字符串, 同时iov_len为3, 故发送ABC。而ptr[1] (数组的第二个元素) 的iov_base指向数字1, 同时iov_len为4, 故发送1234。

13.3 问题探讨

(1) 下列关于MSG_OOB可选项的说法错误的是?

- MSG_OOB指传输Out-of-band数据, 是通过其他路径高速传输数据。 X
- MSG_OOB指通过其他路径高速传输数据, 因此, TCP中设置该选项的数据先到达对方主机。 X
- 设置MSG_OOB使数据先到达对方主机后, 以普通数据的形式和顺序读取。也就是说, 只是提高了传输速度, 接收方无法识别这一点。 X
- MSG_OOB无法脱离TCP的默认数据传输方式。即使设置了MSG_OOB, 也会保持原有传输顺序。该选项只用于要求接收方紧急处理。 ✓

2. 利用readv和writev函数收发数据有何优点? 分别从函数调用次数和I/O缓冲的角度给出说明

通过readv和writev来收发数据，可以减少函数调用次数。并且可以将不同缓冲（数组）中的数据同时进行收发。

此外，还可以根据需要，将缓冲中的数据进行分段读取

3. 通过recv函数验证输入缓冲是否存在数据时（确认后立即返回），如何设置recv函数最后一个参数中的可选项？并说明其他可选项的意义

表13-1 send&recv函数的可选项及含义

可选项（Option）	含 义	send	recv
MSG_OOB	用于传输带外数据（Out-of-band data）	·	·
MSG_PEEK	验证输入缓冲中是否存在接收的数据	·	·
MSG_DONTROUTE	数据传输过程中不参照路由（Routing）表，在本地（Local）网络中寻找目的地	·	·
MSG_DONTWAIT	调用I/O函数时不阻塞，用于使用非阻塞（Non-blocking）I/O	·	·
MSG_WAITALL	防止函数返回，直到接收全部请求的字节数	·	·

通常将MSG_PEEK和MSG_DONTWAIT一起使用，用来检测缓冲中的是
否存在数据

14. 多播和广播

14.1 多播的数据传输方式及流量方面的优点

数据特点

- 多播服务器端针对特定多播组，只发送一次数据
- 即使只发送一次数据，但该组内的所有客户端都会接收数据
- 多播组数可在IP地址范围内任意增加
- 加入特定组即可接收发往该多播组的数据
- **多播组是D类IP地址**

多播是基于UDP完成的
多播的数据包格式与UDP
数据包相同，只是在传递
1个多播数据包时，路由
器会复制该数据包并传递
到多个主机



图14-1 多播路由

多播的作用：多播并没有减少网络上的数据包个数。但是能够减少发送次数。
其他的文件由路由器负责复制并传递到对应的主机。

但不是所有路由器都支持多播，对于不支持多播的路由器，可以使用**隧道技术**
完成多播通信

14.1.1 路由 (routing) 和TTL (Time to Live, 生存时间) 以及加入组的方法

TTL是决定“数据包传输距离”的主要因素。TTL用整数表示，并且没经过一个路由器就减1；当TTL变为0时，该数据包无法再被传递，只能销毁。

TTL的值设置的过大将影响网络流量，过小则会导致无法传递到目标主机

TTL设置代码示例：

```
1 int send_sock;
2 int time_live = 64;
3
4 //设置TTL
5 send_sock = socket(PF_INET, SOCK_DGRAM, 0); //采用UDP协议
6 setsockopt(send_sock, IPPROTO_IP, IP_MULTICAST_TTL,
7             (void*)&time_live, sizeof(time_live)); //相关协议层为
8             IPPROTO_IP, 协议参数为IP_MULTICAST_TTL
9
10 //加入多播组
11 int recv_sock;
12 struct ip_mreq join_addr;
13
14 recv_sock = socket(PF_INET, SOCK_DGRAM, 0);
15
16 join_addr.imr_multiaddr.s_addr = "多播组地址信息";
17 join_addr.imr_interface.s_addr = "加入多播组的主机地址信息";
18 setsockopt(recv_sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
19             (void*)&join_addr, sizeof(join_addr));
20
21 //关于结构体ip_mreq
22 struct ip_mreq
23 {
24     struct in_addr imr_multiaddr; //写入加入的组IP地址
25     struct in_addr imr_interface; //加入该组的套接字所属主
26     //机的IP地址，也可以使用INADDR_ANY
27 }
28
29 struct in_addr{
30     In_addr_t s_addr; //32位IPv4地址
31 };
```

INADDR_ANY: 如果bind绑定的是**INADDR_ANY**, 即表示所有发送到服务器的这个端口,

不管是哪个网卡/哪个IP地址接收到的数据, 都由这个服务端进程进行处理。

14.2 实现多播Sender和Receiver

多播中用发送者和接受者替代服务器端和客户端。

```
1 //sender.c
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <unistd.h>
6 #include <arpa/inet.h>
7 #include <sys/socket.h>
8
9 #define TTL 64
10#define BUF_SIZE 30
11
12 void error_handling(char* message);
13
14 int main(int argc, char* argv[])
15 {
16     int send_sock;
17     struct sockaddr_in mul_addr;
18     int time_live = TTL;
19     FILE* fp;
20     char buf[BUF_SIZE];
21
22     if(argc != 3) //接受三个参数。因为不像服务端可以指定IP地址为INADDR_ANY
23     {
24         printf("Usage: %s <GroupIP> <PORT>\n",
25                argv[0]);
26         exit(1);
27     }
28
29     send_sock = socket(PF_INET, SOCK_DGRAM, 0); //使用
30     UDP协议
31     memset(&mul_addr, 0, sizeof(mul_addr));
32     mul_addr.sin_family = AF_INET;
```

```
31     mul_addr.sin_addr.s_addr = inet_addr(argv[1]);
//multicast IP
32     mul_addr.sin_port = htons(atoi(argv[2])); //Port
33
34     setsockopt(send_sock, IPPROTO_IP,
IP_MULTICAST_TTL, (void*)&time_live,
sizeof(time_live)); //设置TTL
35
36     if((fp = fopen("news.txt", "r")) == NULL)
            error_handling("fopen() error");
37     while(!feof(fp)) //Broadcasting
{
38         fgets(buf, BUF_SIZE, fp);
39         sendto(send_sock, buf, strlen(buf), 0, (struct
40 sockaddr*)&mul_addr, sizeof(mul_addr)); //因为是通过UDP套接
41         字传输数据，因此使用sendto函数。
42         //send(c1nt_sock, buf, sizeof(buf));
43         //write(c1nt_sock, buf, str_len ); tcp的write不
44         需要提供IP地址等
45         sleep(2);
46     }
47     fclose(fp); //关闭打开的文件
48     close(send_sock);
49     return 0;
50 }
51 void error_handling(char* message)
52 {
53     fputs(message, stderr);
54     fputc('\n', stderr);
55     exit(1);
56 }
57
58
59 //receiver.c
60 #include <stdio.h>
61 #include <stdlib.h>
62 #include <string.h>
63 #include <unistd.h>
64 #include <arpa/inet.h>
65 #include <sys/socket.h>
```

```
66
67 #define BUF_SIZE 30
68 void error_handling(char* message);
69
70 int main(int argc, char* argv[])
71 {
72     int recv_sock;
73     int str_len;
74     char buf[BUF_SIZE];
75     struct sockaddr_in adr;
76
77     struct ip_mreq join_adr;
78     if(argc != 3 )
79     {
80         printf("Usage: %s <GroupIP> <PORT>\n",
81               argv[0]);
82         exit(1);
83     }
84
85     recv_sock = socket(PF_INET, SOCK_DGRAM, 0);
86     memset(&adr, 0, sizeof(adr));
87     adr.sin_family = AF_INET; //IPv4地址
88     adr.sin_addr.s_addr = htonl(INADDR_ANY); //设定主机
的任一IP都可以接受
89     adr.sin_port = htons(atoi(argv[2])); //设置主机的端口
号
90
91     if(bind(recv_sock, (struct sockaddr*)&adr,
92             sizeof(adr)) == -1)
93         error_handling("bind() error"); //创建有链接的
UDP
94
95     join_adr.imr_multiaddr.s_addr =
96     inet_addr(argv[1]);
97     join_adr.imr_interface.s_addr = htonl(INADDR_ANY);
//设定多播组的地址以及待加入多播组的IP地址
98
99     setsockopt(recv_sock, IPPROTO_IP,
100     IP_ADD_MEMBERSHIP, (void*)&join_adr,
101     sizeof(join_adr));
102     while(1)
```

```

98  {
99      str_len = recvfrom(recv_sock, buf, BUF_SIZE -
100     1, 0, NULL, 0);
101     if(str_len < 0) break;
102     buf[str_len] = 0;
103     fputs(buf, stdout);
104 }
105 close(recv_sock);
106 return 0;
107 }
108
109 void error_handling(char* message)
110 {
111     fputs(message, stderr);
112     fputc('\n', stderr);
113     exit(1);
114 }

```

代码分析:

1. 要求sender和receiver的传入参数一致。即sender要给定传递给的多播组IP地址，而receiver需要设置需要加入的多播组IP地址。

TCP和UDP的数据传输

1. UDP套接字发送数据是采用 `sendto()`，因为没有建立连接，所以需要加上接收方的IP地址，端口号等信息
2. TCP可以通过 `read/write`, `send()/recv()` 以及 `writev/readv`

14.3 广播

广播vs多播：多播即使在跨越不同网络的情况下，只要加入多播组就能接收数据。
相反，广播只能在同一网络中的主机传输数据

14.3.1 广播的理解及实现方法

广播是向同一网络中的所有主机传输数据的方法，也是基于UDP完成的。

根据传输数据时使用的IP地址形式，广播可分为

1. 直接广播
2. 本地广播

二者在代码实现上的差别主要在于IP地址。直接广播的IP地址中除了网络地址外，其余主机地址全部设置为1。例如，希望向网络地址192.12.34中的所有主机传输数据时，可以向192.12.34.255传输。换言之，可以采用直接广播的方式向特定区域内所有主机传输数据。

反之，本地广播中使用的IP地址限定为255.255.255.255。例如，192.32.24网络中的主机向255.255.255.255传输数据时，数据将传递到192.32.24网络中的所有主机。

那么，应当如何实现Sender和Receiver呢？实际上，如果不仔细观察广播示例中通信时使用的IP地址，则很难与UDP示例进行区分。也就是说，数据通信中使用的IP地址是与UDP示例的唯一区别。默认生成的套接字会阻止广播，因此，只需通过如下代码更改默认设置。

```
int send_sock;
int bcast = 1; // 对变量进行初始化以将 SO_BROADCAST 选项信息改为 1。
. . .
send_sock = socket(PF_INET, SOCK_DGRAM, 0);
. . .
setsockopt(send_sock, SOL_SOCKET, SO_BROADCAST, (void*) &bcast, sizeof(bcast));
. . .
```

调用setsockopt函数，将SO_BROADCAST选项设置为bcast变量中的值1。这意味着可以进行数据广播。当然，上述套接字选项只需在Sender中更改，Receiver的实现不需要该过程。

```
1 //sender.c
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <unistd.h>
6 #include <arpa/inet.h>
7 #include <sys/socket.h>
8
9 #define TTL 64 //因为没有经过理由转发就不需要TTL
10#define BUF_SIZE 30
11
12 void error_handling(char* message);
13
14 int main(int argc, char* argv[])
15 {
16     int send_sock;
17     struct sockaddr_in broad_addr;
18     int brd_tag = 1;
19     int time_live = TTL;
20     FILE* fp;
21     char buf[BUF_SIZE];
22
23     if(argc != 3) //接受三个参数。因为不像服务端可以指定IP地址为INADDR_ANY
24     {
```

```
25     printf("Usage: %s <GroupIP> <PORT>\n",
26             argv[0]);
27 }
28
29     send_sock = socket(PF_INET, SOCK_DGRAM, 0); //使用
30     UDP协议
31     memset(&broad_addr, 0, sizeof(broad_addr));
32     broad_addr.sin_family = AF_INET;
33     broad_addr.sin_addr.s_addr = inet_addr(argv[1]);
34     //broadcast IP
35     broad_addr.sin_port = htons(atoi(argv[2])); //Port
36
37     setsockopt(send_sock, SOL_SOCKET, SO_BROADCAST,
38     (void*)&brd_tag, sizeof(brd_tag)); //但是需要打开广播
39     //setsockopt(send_sock, IPPROTO_IP,
40     IP_MULTICAST_TTL, (void*)&time_live,
41     sizeof(time_live)); //不需要设置TTL
42
43     if((fp = fopen("news.txt", "r")) == NULL)
44         error_handling("fopen() error");
45     while(!feof(fp)) //Broadcasting
46     {
47         fgets(buf, BUF_SIZE, fp);
48         sendto(send_sock, buf, strlen(buf), 0, (struct
49         sockaddr*)&broad_addr, sizeof(broad_addr)); //因为是通过
50         UDP套接字传输数据，因此使用sendto函数。
51         //send(cInt_sock, buf, sizeof(buf));
52         //write(cInt_sock, buf, str_len ); tcp的write不
53         需要提供IP地址等
54         sleep(2);
55     }
56     fclose(fp); //关闭打开的文件
57     close(send_sock);
58     return 0;
59 }
60
61 void error_handling(char* message)
62 {
63     fputs(message, stderr);
64     fputc('\n', stderr);
```

```
57     exit(1);
58 }
59
60
61 //receiver.c
62 #include <stdio.h>
63 #include <stdlib.h>
64 #include <string.h>
65 #include <unistd.h>
66 #include <arpa/inet.h>
67 #include <sys/socket.h>
68
69 #define BUF_SIZE 30
70 void error_handling(char* message);
71
72 int main(int argc, char* argv[])
73 {
74     int recv_sock;
75     int str_len;
76     char buf[BUF_SIZE];
77     struct sockaddr_in adr;
78
79     if(argc != 2)
80     {
81         printf("Usage: %s <PORT>\n", argv[0]);
82         exit(1);
83     }
84
85     recv_sock = socket(PF_INET, SOCK_DGRAM, 0);
86     memset(&adr, 0, sizeof(adr));
87     adr.sin_family = AF_INET; //IPv4地址
88     adr.sin_addr.s_addr = htonl(INADDR_ANY); //设定主机
的任一IP都可以接受
89     adr.sin_port = htons(atoi(argv[1])); //设置主机的端口
号
90
91     if(bind(recv_sock, (struct sockaddr*)&adr,
92             sizeof(adr)) == -1)
93         error_handling("bind() error"); //创建有链接的
UDP
```

```

94     //setsockopt(recv_sock, IPPROTO_IP,
95     IP_ADD_MEMBERSHIP, (void*)&join_addr,
96     sizeof(join_addr)); //不需要加入多播组
97     while(1)
98     {
99         str_len = recvfrom(recv_sock, buf, BUF_SIZE -
100        1, 0, NULL, 0);
101        if(str_len < 0) break;
102        buf[str_len] = 0;
103        fputs(buf, stdout);
104    }
105 }
106
107 void error_handling(char* message)
108 {
109     fputs(message, stderr);
110     fputc('\n', stderr);
111     exit(1);
112 }
```

代码运行结果：

1. 本地广播 `./broadsender 255.255.255.255 9191`
2. 直接广播 `./broadsender 172.30.15.255 9191`

这里的直接广播地址是通过ifconfig得到的

```
[xiaotang@VM-0-5-centos src]$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.30.0.5 netmask 255.255.240.0 broadcast 172.30.15.255
        ether 52:54:00:cf:aa:5e txqueuelen 1000 (Ethernet)
        RX packets 9362485 bytes 1000967986 (954.5 MiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 9151508 bytes 1594689120 (1.4 GiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
        inet6 ::1 prefixlen 128 scopeid 0x10<host>
            loop txqueuelen 1000 (Local Loopback)
            RX packets 13511 bytes 911367 (890.0 KiB)
            RX errors 0 dropped 0 overruns 0 frame 0
```

14.4 问题探讨

1. TTL的含义是什么？请从路由器的角度说明较大的TTL值与较小的TTL值之间的区别及问题

TTL指的是数据包生存周期，决定数据包的传输距离。当数据包经过一个一个路由器转发会TTL减1，当TTL等于0时则销毁数据包

在数据传输过程中，过大的TTL值会导致网络中数据包数量增加，可能会造成网路阻塞

而较小的TTL值又可能会导致数据包无法到达目的主机

2. 多播和广播的异同点是什么？

多播和广播的相同点是，两者都是以UDP形式传输数据。一次传输数据，可以向两个以上主机传送数据。但传送的范围是不同的：广播是对局域网的广播；而多播是对网络注册机器的多播

3. 多播也对网络流量有利，请比较TCP数据交换方式解释其原因。

多播数据在路由器进行复制。因此，即使主机数量很多，如果各主机存在的相同路径，也可以通过一次数据传输到多台主机上。但TCP无论路径如何，都要根据主机数量进行数据传输。

15. 套接字和标准I/O

15.1 使用标准I/O函数的两个优点

1. 标准I/O函数具有良好的移植性
2. 标准I/O函数可以利用缓冲提高性能

接下来讨论标准I/O函数的第二个优点。使用标准I/O函数时会得到额外的缓冲支持。这种表达方式也许会带来一些混乱，因为之前讲过，创建套接字时操作系统会准备I/O缓冲。造成更大混乱之前，先说明这两种缓冲之间的关系。创建套接字时，操作系统将生成用于I/O的缓冲。此缓冲在执行TCP协议时发挥着非常重要的作用。此时若使用标准I/O函数，将得到额外的另一缓冲的支持，如图15-1所示。

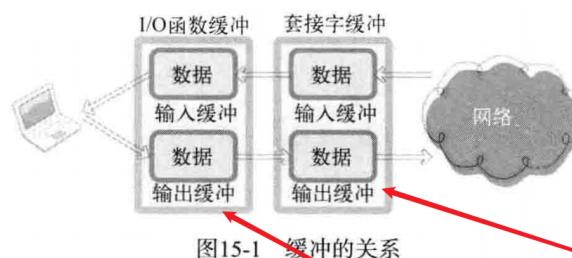


图15-1 缓冲的关系

从图15-1中可以看到，使用标准I/O函数传输数据时，经过2个缓冲。例如，通过 fputs 函数传输字符串“Hello”时，首先将数据传递到标准I/O函数的缓冲。然后数据将移动到套接字输出缓冲，最后将字符串发送到对方主机。

设置缓冲的用途：

1. 设置缓冲的主要目的是为了提高性能（针对标准I/O函数缓冲而言）

标准I/O函数为了提高性能，内部提供额外的缓冲。因此，若不调用 `fflush` 函数则无法保证立即将数据传输到客户端

```
1 while(!feof(readfp))
2 {
3     fgets(message, BUF_SIZE, readfp);
4     fputs(message, writefp);
5     fflush(writefp);
6 }
```

2. 套接字的缓冲主要是为了TCP协议而设立的，为的是能够实现再次重传。因此，在套接字的输出缓冲中保存了已经发送的数据

来的性能差异越大。可以通过如下两种角度说明性能的提高。

- 传输的数据量
- 数据向输出缓冲移动的次数

①

②

比较1个字节的数据发送10次（10个数据包）的情况和累计10个字节发送1次的情况。发送数据时使用的数据包中含有头信息。头信息与数据大小无关，是按照一定的格式填入的，即使假设该头信息占用40个字节（实际更大），需要传递的数据量也存在较大差别。

- 1个字节 10次 $40 \times 10 = 400$ 字节
- 10个字节 1次 $40 \times 1 = 40$ 字节 ✓

另外，为了发送数据，向套接字输出缓冲移动数据也会消耗不少时间。但这同样与移动次数有关。1个字节数据共移动10次花费的时间将近10个字节数据移动1次花费时间的10倍。✗

15.1.1 标准I/O函数的几个缺点

如果就此结束说明，各位可能认为标准I/O函数只有优点。其实它同样有缺点，整理如下。

- 不容易进行双向通信。
- 有时可能频繁调用 `fflush` 函数。
- 需要以FILE结构体指针的形式返回文件描述符。

FILE* fp1; //FILE结构体指针

假设各位已掌握了C语言中的绝大部分文件I/O相关知识。打开文件时，如果希望同时进行读写操作，则应以r+、w+、a+模式打开。但因为缓冲的缘故，每次切换读写工作状态时应调用 `fflush` 函数。这也将影响基于缓冲的性能提高。而且，为了使用标准I/O函数，需要FILE结构体指针（以下简称“FILE指针”）。而创建套接字时默认返回文件描述符，因此需要将文件描述符转化为FILE指针。若各位难以分清FILE指针和文件描述符，可以通过上述 `syscpy.c` 和 `stdcpy.c` 示例加以区分。

15.1.2 使用标准I/O函数

对于创建套接字时返回的文件描述符 `fd`，需要通过 `fdopen()` 函数将其转换为 FILE结构体指针 才能使用标准I/O函数

```
1 #include <stdio.h>
2
3 FILE* fdopen(int filedes, const char* mode); //成功时返回
        //转换的FILE结构体指针，失败时返回NULL
```

```
4 //fildes: 需要转换的文件描述符
5 //mode: 将要创建的FILE结构体指针的模式信息
6 // 1. "r"读模式
7 // 2. "w"写模式
8
9 //fdopen代码示例
10 #include <stdio.h>
11 #include <fcntl.h>
12
13 int main(void)
14 {
15     FILE *fp;
16     int fd = open("data.dat", O_WRONLY | O_CREAT |
17 O_TRUNC); //创建文件描述符
17     if(fd == -1)
18     {
19         fputs("file open error", stdout);
20         return -1;
21     }
22     fp = fdopen(fd, "w"); //将文件描述符fd通过fdopen转换为
23 //FILE*,并且打开模式为w
24     fputs("Network C programming \n", fp); //通过标准I/O的
25 //fputs写入字符串"Network C programming"
26
27     fclose(fp);
28 }
```

利用fileno函数转换为文件描述符

```
1 #include <stdio.h>
2
3 int fileno(FILE* stream); //成功时返回转换后的文件描述符, 失败
4 时返回-1
```

❖ todes.c

```
1. #include <stdio.h>
2. #include <fcntl.h>
3.
4. int main(void)
5. {
6.     FILE *fp; //文件描述符转为FILE*
7.     int fd=open("data.dat", O_WRONLY|O_CREAT|O_TRUNC); 文件描述符
8.     if(fd== -1)
9.     {
10.         fputs("file open error", stdout);
11.         return -1;
12.     }
13.
14.     printf("First file descriptor: %d \n", fd);
15.     fp=fdopen(fd, "w"); 通过fdopen函数将文件描述符转换为FILE*
16.     fputs("TCP/IP SOCKET PROGRAMMING \n", fp); 通过新转换的FILE指针写入字符串
17.     printf("Second file descriptor: %d \n", fileno(fp)); 将fp通过fileno又转换为文件描述符
18.     fclose(fp);
19.     return 0;
20. }
```

15.2 基于套接字的标准I/O函数使用

在第四章的基础上，将文件描述符通过 `fdopen` 函数转为 `FILE*` 并使用标准I/O函数

1. echo_server.c的两种实现方式

```
for(i=0; i<5; i++) 至多连接五个client
{
    clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_sz); accept: 接受客户端的请求，并自动生成I/O套接字
    if(clnt_sock== -1)
        error_handling("accept() error");
    else
        printf("Connected client %d \n", i+1);

    while((str_len=read(clnt_sock, message, BUF_SIZE))!=0) 对每个client，自动读取并反馈字符串回去
        write(clnt_sock, message, str_len);

    close(clnt_sock); 关闭当前连接
}
close(serv_sock); 关闭服务器
return 0;
}

clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_sz); 通过将clnt_sock转换为readfp和writefp
if(clnt_sock== -1)
    error_handling("accept() error");
else
    printf("Connected client %d \n", i+1);

readfp=fdopen(clnt_sock, "r");
writefp=fdopen(clnt_sock, "w"); 进而使用标准I/O函数fgets()和fputs()
while(!feof(readfp))
{
    fgets(message, BUF_SIZE, readfp);
    fputs(message, writefp);
    fflush(writefp);
}
fclose(readfp);
fclose(writefp);
```

标准I/O函数为了提高性能，内部提供额外的缓冲。因此，若不调用 `fflush` 函数则无法保证立即将数据传输到客户端

2. echo_client.c的两种实现方式

```
readfp=fdopen(sock, "r");
writefp=fdopen(sock, "w");
while(1)
{
    fputs("Input message(Q to quit): ", stdout);
    fgets(message, BUF_SIZE, stdin);
    if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
        break;

    fputs(message, writefp);
    fflush(writefp);
    fgets(message, BUF_SIZE, readfp); 使用标准I/O的输入输出函数
    printf("Message from server: %s", message);
}
```

通过fflush及时更新写入的文件内容

15.3 问题探讨

1. 标准I/O函数的两个优点？为什么具有这两个优点
 - 基于ANSIX标准具有良好的一致性
 - 可以利用缓冲提高性能
2. 标准IO中，“调用fputs函数传输数据时，调用后应立即开始发送！”，为何这种想法是错误的？为了达到这种效果应添加哪些处理过程？

通过标准输出函数的传输的数据不直接通过套接字的输出缓冲区发送，而是保存在标准输出函数的缓冲中，然后再用 **fflush** 函数进行输出。因此，即使调用 **fputs** 函数，也不能立即发送数据。如果想保障数据传输的时效性，必须经过 **fflush** 函数的调用过程

16. 关于I/O流分离的其他内容

16.1 分离I/O

流：指数据流动，但通常可以比喻为“以数据收发为目的的一种桥梁”

分离I/O的两种方法：

1. 通过创建多个进程，分割数据收发流程。另一个好处：可以提高频繁交换数据的程序性能

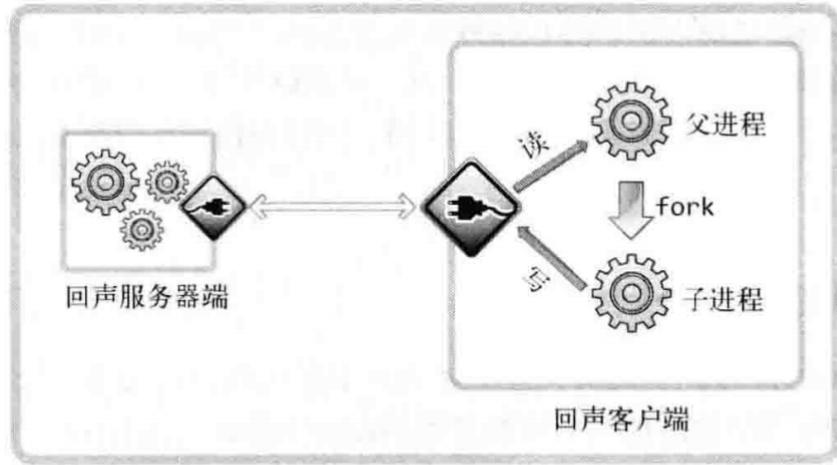


图10-5 回声客户端I/O分割模型

目的:

- 通过分开输入过程和输出过程降低实现难度
 - 与输入无关的输出操作可以提高速度
2. 通过对 `fdopen` 函数的调用，创建读模式和写模式的FILE指针，即分离了输入工具和输出工具

目的:

- 为了将FILE指针按照读模式和写模式加以区分
- 可以通过区分读写模式降低实现难度
- 通过区分I/O缓冲提高缓冲性能

流分离带来的EOF问题：参考第七章：优雅地断开套接字连接中提到的**半关闭**:
`shutdown(sock, SHUT_WR)`

但是对于 `FILE指针` 的输出模式

```

30.     readfp=fopen(c Clint_sock, "r");
31.     writefp=fopen(c Clint_sock, "w");
32. 
33.     fputs("FROM SERVER: Hi~ client? \n", writefp);
34.     fputs("I love all of the world \n", writefp);
35.     fputs("You are awesome! \n", writefp);
36.     fflush(writefp);
37.             先将服务器端的三串字符发送，并传递给客户端，并且关闭writefp
38.     fclose(writefp); 此时发送一个eof给客户端。
39.     fgets(buf, sizeof(buf), readfp); 这时服务器端的读操作理论上可行，等待客户端的发送内容
40.     fputs(buf, stdout);
41.     fclose(readfp);
42.     return 0;
43. }
```

通过`fdopen`将文件描述符转为FILE*之后
I/O分离服务端测试代码

```

```
19. connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
20. readfp=fopen(sock, "r"); 客户端测试代码
21. writefp=fopen(sock, "w");
22.
23. while(1)
24. {
25. if(fgets(buf, sizeof(buf), readfp)==NULL)当接收到服务器端发送过来的EOF之后
26. break; 停止接受，并跳出while循环
27. fputs(buf, stdout);
28. fflush(stdout);
29. }
30.
31. fputs("FROM CLIENT: Thank you! \n", writefp); 客户端发送消息给服务器端并断开连接
32. fflush(writefp);
33. fclose(writefp); fclose(readfp);
34. return 0;
35. }

```

在上述代码中，当服务器端关闭输出缓冲writefp的时候，这时候，**套接字创建的tcp连接已经关闭。而不是半关闭**

因此，本章需要实现**文件描述符的复制和半关闭**，即针对**fdopen**函数调用时生成的**FILE**指针进行半关闭操作

## 16.2 文件描述符的复制和半关闭

### + 终止“流”时无法半关闭的原因

图16-1描述的是sep\_serv.c示例中的2个FILE指针、文件描述符及套接字之间的关系。

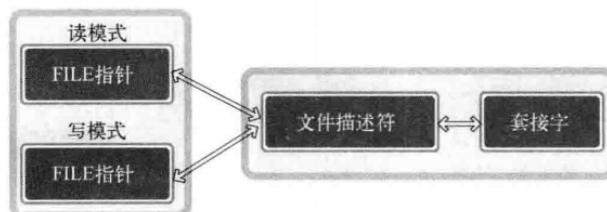


图16-1 FILE指针的关系

从图16-1中可以看到，示例sep\_serv.c中的读模式FILE指针和写模式FILE指针都是基于同一文件描述符创建的。因此，针对任意一个FILE指针调用fclose函数时都会关闭文件描述符，也就终止套接字，如图16-2所示。

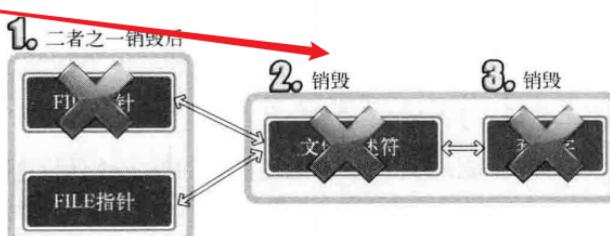


图16-2 调用fclose函数的结果

从图16-2中可以看到，销毁套接字时再也无法进行数据交换。那如何进入可以输入但无法输出的半关闭状态呢？其实很简单。如图16-3所示，创建FILE指针前先复制文件描述符即可。

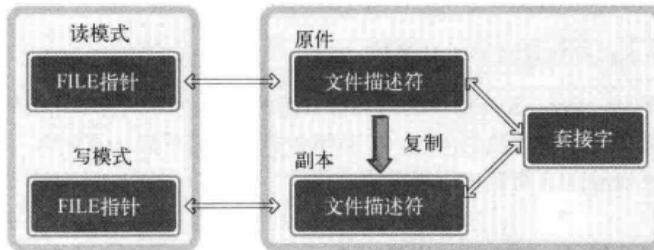


图16-3 半关闭模型1

如图16-3所示，复制后另外创建1个文件描述符，然后利用各自的文件描述符生成读模式FILE指针和写模式FILE指针。这就为半关闭准备好了环境，因为套接字和文件描述符之间具有如下关系：

“销毁所有文件描述符后才能销毁套接字。”

也就是说，针对写模式FILE指针调用fclose函数时，只能销毁与该FILE指针相关的文件描述符，无法销毁套接字（参考图16-4）。

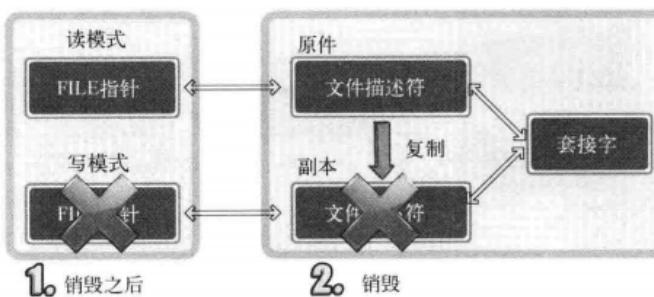


图16-4 半关闭模型2

如图16-4所示，调用fclose函数后还剩1个文件描述符，因此没有销毁套接字。那此时的状态是否为半关闭状态？不是！图16-3中讲过，只是准备好了半关闭环境。要进入真正的半关闭状态需要特殊处理。

### 16.2.1 复制文件描述符

这里的文件描述符的复制与fork函数中进行的复制有所区别，调用fork函数时将复制整个进程。因此同一进程内不能同时有原件和副本

```
1 #include <unistd.h>
2
3 int dup(int fildes);
4 int dup2(int fildes, int fildes2); //成功时返回复制的文件描述符，失败时返回-1
5 /*
6 fildes: 需要复制的文件描述符
7 fildes2: 明确指定的文件描述符整数值，即指定复制出的文件描述符的数值。要求大于0且小于进程能生成的最大文件描述符值
8 */
9
10 //代码示例
11 #include <stdio.h>
```

```

12 #include <unistd.h>
13
14 int main(int argc, char* argv[])
15 {
16 int cfd1, cfd2;
17 char str1[] = "Hi~ \n";
18 char str2[] = "It's a nice day~ \n";
19
20 cfd1 = dup(1);
21 cfd2 = dup2(cfd1, 7);
22
23 printf("fd1 = %d, fd2 = %d \n", cfd1, cfd2);
24 write(cfd1, str1, sizeof(str1));
25 write(cfd2, str2, sizeof(str2));
26
27 close(cfd1);
28 close(cfd2);
29
30 write(1, str1, sizeof(str1));
31 close(1);
32 write(1, str2, sizeof(str2));
33
34 return 0;
35 }

```

**代码说明**

- 第10、11行：第10行调用dup函数复制了文件描述符1。第11行调用dup2函数再次复制了文件描述符，并指定描述符整数值为7。
- 第14、15行：利用复制出的文件描述符进行输出。通过该输出结果可以验证是否进行了实际复制。
- 第17~19行：终止复制的文件描述符。但仍有1个描述符，因此可以进行输出。可以从第19行得到验证。close(cfd1, cfd2)
- 第20、21行：第20行终止最后的文件描述符，因此无法完成第21行的输出。

❖ 运行结果：dup.c

```

root@my_linux:/tcpip# gcc dup.c -o dup
root@my_linux:/tcpip# ./dup
fd1=3, fd2=7
Hi~
It's nice day~
Hi~

```

## 16.2.2 复制文件描述符后“流”的分离

之前，在将 `fdopen()` 建立的读写 FILE 指针关闭其一之后，这时候便会发送 `EOF` 使得另一个指针无法正常工作。因为文件描述符只有一个，被关闭之后建立的套接字也随之关闭。

因此需要通过 `dup` 或者 `dup2` 函数将文件描述符进行复制

进而实现文件描述符或 FILE\* 的半关闭

**sep\_serv2.c 代码示例：**

```
24.
25. readfp=fdopen(clnt_sock, "r");
26. writefp=fdopen(dup(clnt_sock), "w");
27.
28. fputs("FROM SERVER: Hi~ client? \n", writefp); *调用shutdown函数时,
29. fputs("I love all of the world \n", writefp); 无论复制出多少文件描述
30. fputs("You are awesome! \n", writefp); 符都进入半关闭状态。
31. fflush(writefp); 同时传入EOF
32.
33. shutdown(fileno(writefp), SHUT_WR);
34. fclose(writefp);
35.
```

• 第33行：针对 fileno 函数返回的文件描述符调用 shutdown 函数。因此，服务器端进入半关闭状态，并向客户端发送 EOF。这一行就是之前所说的发送 EOF 的方法。调用 shutdown 函数时，无论复制出多少文件描述符都进入半关闭状态，同时传递 EOF。

## 16.3 问题探讨

### 1. 下列关于 FILE 结构体指针和文件描述符的说法错误的是？

答：以下加粗内容代表说法正确。

1. 与 FILE 结构体指针相同，文件描述符也分输入描述符和输出描述符
2. 复制文件描述符时将生成相同值的描述符，可以通过这 2 个描述符进行 I/O
3. **可以利用创建套接字时返回的文件描述符进行 I/O，也可以不通过文件描述符，直接通过 FILE 结构体指针完成**
4. **可以从文件描述符生成 FILE 结构体指针，而且可以利用这种 FILE 结构体指针进行套接字 I/O**
5. 若文件描述符为读模式，则基于该描述符生成的 FILE 结构体指针同样是读模式；若文件描述符为写模式，则基于该描述符生成的 FILE 结构体指针同样也是写模式

### 2. EOF 的发送相关描述中错误的是？

答：以下加粗内容代表说法正确。

1. 终止文件描述符时发送 EOF
2. **即使未完成终止文件描述符，关闭输出流时也会发送 EOF**
3. 如果复制文件描述符，则包括复制的文件描述符在内，所有文件描述符都终止时才会发送 EOF
4. **即使复制文件描述符，也可以通过调用 shutdown 函数进入半关闭状态并发送 EOF**

## 17. 优于 select 的 epoll

实现I/O复用的传统方法有 **select** 函数 和 **poll** 函数，但是各种原因导致这些方法无法得到令人满意的性能。因此有了 **linux** 下的 **epoll**, **BSD** 的 **kqueue**, **Solaris** 的 **/dev/poll** 和 **windows** 下的 **IOCP** 等复用技术

**select** 方式并不适合以 **web** 服务器端开发 为主的现代开发环境，所以要学习 **linux** 平台下的 **epoll**

## 17.1 基于**select**的I/O复用技术速度慢的原因

### 17.1.1 **select**的缺点

- 调用 **select** 函数后常见的针对所有文件描述符的循环语句
- 每次调用 **select** 函数时都需要向该函数传递监视对象信息

```
while(1)
{
 temps = reads; //将准备好的fd_set变量的内容复制到temps变量
 //因为调用select函数后，除发生变化的文件描述符对应位外，剩下的所有位都将初始化为0
 //因此为了记住初始值，必须经过这种复制过程。这也是select函数的通用方法
 timeout.tv_sec = 5;
 timeout.tv_usec = 0;
 result = select(1, &temps, 0, 0, &timeout);
 if(result == -1)
 {puts("select() error"); break;}
 else if(result == 0) puts("time out!");
 else
 {
 if(FD_ISSET(0, &temps))
 {

 str_len = read(0, buf, BUF_SIZE);
 buf[str_len] = 0;
 printf("message from console: %s", buf);
 }
 }
}
```

上述两点可以从第12章示例 **echo\_selectserv.c** 的第45、49行及第54行代码得到确认。调用 **select** 函数后，并不是把发生变化的文件描述符单独集中到一起，而是通过观察作为监视对象的 **fd\_set** 变量的变化，找出发生变化的文件描述符（示例 **echo\_selectserv.c** 的第54、56行），因此无法避免针对所有监视对象的循环语句。而且，作为监视对象的 **fd\_set** 变量会发生变化，所以调用 **select** 函数前应 **复制并保存原有信息**（参考 **echo\_selectserv.c** 的第45行），并在每次调用 **select** 函数时传递新的监视对象信息。**占用内存**

只看代码的话，循环体是对提高性能的大障碍，而**更大的障碍是每次传递监视对象的信息**

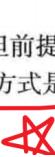
应用程序向操作系统传递数据时，将对程序造成很大的负担。而且无法通过代码优化解决。因此将造成性能上的致命弱点

“那为何需要把监视对象信息传递给操作系统呢？”

有些函数不需要操作系统的帮助就能完成功能，而有些则必须借助于操作系统。假设各位定义了四则运算相关函数，此时无需操作系统的帮助。但select函数与文件描述符有关，更准确地说，是监视套接字变化的函数。而套接字是由操作系统管理的，所以select函数绝对需要借助于操作系统才能完成功能。select函数的这一缺点可以通过如下方式弥补：

“仅向操作系统传递1次监视对象，监视范围或内容发生变化时只通知发生变化的事项。”

这样就无需每次调用select函数时都向操作系统传递监视对象信息，但前提是操作系统支持这种处理方式（每种操作系统支持的程度和方式存在差异）。Linux的支持方式是epoll，Windows的支持方式是IOCP。



## 17.1.2 select函数的优点

epoll方式只在linux下提供支持，即改进的I/O复用模式不具有兼容性。而大多数操作系统都支持select函数。

## 17.2 实现epoll时必要的函数和结构体

能够克服select函数缺点的epoll函数具有如下优点（与select函数的缺点相反）

1. 无需编写以监视状态变化为目的的针对所有文件描述符的循环语句
2. 调用对应于epoll函数的 epoll\_wait 函数时无需每次传递监视对象信息

### epoll服务器端实现中需要的三个函数

1. epoll\_create:创建保存epoll文件描述符的空间
2. epoll\_ctl:向空间注册并销毁文件描述符
3. epoll\_wait:与select函数类型，等待文件描述符发生变化

### 17.2.1 select VS epoll

1. select方式中为了保存监视对象文件描述符，直接通过 fd\_set 变量来设置文件描述符。而 epoll 方式下由操作系统负责保存监视对象文件描述符。因此需要通过 epoll\_create 函数向操作系统请求创建 保存文件描述符的空间
2. **为了添加和删除监视对象文件描述符**
  - select通过 FD\_SET, FD\_CLR 函数
  - 在epoll中，通过epoll\_ctl函数请求操作系统完成
3. select通过 select 函数等待文件描述符的变化，而epoll中调用 epoll\_wait 函数
4. select通过 FD\_ISSET 查看fd\_set变量查看监视对象的状态变化。而epoll方式中通过 epoll\_event 将发生变化的文件描述符单独集中在一起

```
1 struct epoll_event
2 {
```

```

3 __uint32_t events;
4 epoll_data_t data;
5 }
6
7 typedef union epoll_data
8 {
9 void* ptr;
10 int fd;
11 __uint32_t u32;
12 __uint64_t u64;
13 } epoll_data_t;
14 //声明足够大的epoll_event结构体数组后，传递给epoll_wait函数时，发生变化的文件描述符信息将被填入该数组。因此无需像select函数那样针对所有文件描述符进行循环

```

## 17.2.2 epoll\_create

注：需要linux内核2.6以上

`uname -srM`查看内核版本

`cat /proc/sys/kernel/osrelease`

```

1 #include <sys/epoll.h>
2
3 int epoll_create(int size); //成功时返回epoll文件描述符，失败时返回-1
4 // size: epoll实例的大小，决定epoll例程的大小，但是该值只是向操作系统提的建议。
5 // 因此，size并非用来决定epoll例程的大小，仅供操作系统参考
6 // 实际上，linux2.6.8之后的内核将完全忽略传入的size参数，并根据情况调整epoll实例的大小

```

通过`epoll_create`函数创建的文件描述符保存空间称为"epoll例程"

epoll\_create函数创建的资源与套接字相同，也由操作系统管理。因此，该函数和创建套接字的情况相同，也会返回文件描述符。也就是说，该函数返回的文件描述符主要用于区分epoll例程。需要终止时，与其他文件描述符相同，也要调用`close`函数。  
→ epoll例程的标识，通过`close`关闭

## 17.2.3 epoll\_ctl

生成epoll例程后，应通过`epoll_ctl`函数在其内部注册监视对象文件描述符

```
1 #include <sys/epoll.h>
```

```

2
3 int epoll_ctl(int epfd, int op, int fd, struct
4 epoll_event* event); //成功时返回0, 失败时返回-1
5 /*
6 epfd:用于注册监视对象的epoll例程的文件描述符
7 op:用于指定监视对象的添加、删除或更改等操作
8 - EPOLL_CTL_ADD:将文件描述符注册到epoll例程
9 - EPOLL_CTL_DEL:从epoll例程中删除文件描述符
10 - EPOLL_CTL_MOD:更改注册的文件描述符的关注事件发生情况
11 fd:需要注册的监视对象文件描述符
12 event:监视对象的事件类型, 从监视对象中删除文件描述符时, 不需要监视
13 类型(事件信息)
14 */
15
16 //函数调用
17 epoll_ctl(A, EPOLL_CTL_ADD, B, C); //向epoll例程A中注册文
18 件描述符B, 主要目的是监视参数C中的事件
19 epoll_ctl(A, EPOLL_CTL_DEL, B, NULL); //从epoll例程A中删
20 除文件描述符B

```

下面讲解各位不太熟悉的epoll\_ctl函数的第四个参数, 其类型是之前将过的epoll\_event结构体指针。

“啊? 不是说epoll\_event用于保存发生变化的(发生事件)的文件描述符吗?”

当然! 如前所述, epoll\_event结构体用于保存发生事件的文件描述符集合。但也可以在epoll例程中注册文件描述符时, 用于注册关注的事件。函数中epoll\_event结构体的定义并不显眼, 因此通过调用语句说明该结构体在epoll\_ctl函数中的应用。

```

1 //epoll_event
2 struct epoll_event
3 {
4 __uint32_t events;
5 epoll_data_t data;
6 }
7 typedef union epoll_data
8 {
9 void* ptr;
10 int fd;
11 __uint32_t u32;
12 __uint64_t u64;
13 } epoll_data_t;
14

```

```
15 //代码示例:
16 struct epoll_event event;
17 event.events = EPOLLIN; //发生需要读取数据的情况时
18 event.data.fd = sockfd;
19 epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &event);
```

### epoll\_event的成员events可以保存的常量及所指的事件类型

1. EPOLLIN:需要读取数据的情况
2. EPOLLOUT:输出缓冲为空, 可以立即发送数据的情况
3. EPOLLPRI:收到OOB数据的情况 (OOB紧急信息)
4. EPOLLRDHUP:断开连接或半关闭的情况, 这在边缘触发方式下非常有用
5. EPOLLERR:发生错误的情况
6. EPOLLET:以边缘触发的方式得到事件通知
7. EPOLLONESHOT:发生一次事件后, 相应文件描述符不再收到事件通知, 因此需要向epoll\_ctl函数的第二个参数传递EPOLL\_CTL\_MOD,再次设置事件可以通过位或运算同时传递多个参数

## 17.2.4 epoll\_wait

与select机制中select函数对应的 `epoll_wait` 函数, epoll相关函数中默认最后调用该函数

```
1 #include <sys/epoll.h>
2
3 int epoll_wait(int epfd, struct epoll_event* events,
4 int maxevents, int timeout); //成功时返回发生事件的文件描述符数, 失败时返回-1
5 /*
6 * epfd:表示事件发生监视范围的epoll例程的文件描述符
7 * events:保存发生事件的文件描述符集合的结构体地址值
8 * maxevents:第二个参数中可以保存的最大事件数
9 * timeout:以1/1000秒为单位的等待时间, 传递-1时, 一直等待直到发生事件
10 */
11 //调用格式
12 int event_cnt;
13 struct epoll_event* ep_events;
14
15 ep_events = malloc(sizeof(struct
16 epoll_event)*EPOLL_SIZE); //宏常量EPOLL_SIZE
17 //申请EPOLL_SIZE个epoll_event的内存空间
```

```
17 event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE,
18 -1);
19 //调用函数后，返回发生事件的文件描述符数，同时在第二参数指向的缓冲
20 中保存发生事件的文件描述符集合。因此，无需像select那样插入针对所
21 有文件描述符的循环
```

## 17.3 基于epoll的回声服务器端

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <arpa/inet.h>
6 #include <sys/socket.h>
7 #include <sys epoll.h>
8
9 #define BUF_SIZE 100
10#define EPOLL_SIZE 50
11
12 void error_handling(char* message);
13
14 int main(int argc, char* argv[])
15 {
16 int serv_sock, clnt_sock;
17 struct sockaddr_in serv_addr, clnt_addr;
18 socklen_t adr_sz;
19 int str_len, i;
20 char buf[BUF_SIZE];
21
22 struct epoll_event* ep_events; //创建一个ep_event数组
23 struct epoll_event event;
24
25 int epfd, event_cnt; //epfd是通过epoll_create创建的
26 //epoll例程
27
28 if(argc!=2)
29 {
30 printf("Usage: %s <port> \n", argv[0]);
31 exit(1);
32 }
```

```
33 //创建sock套接字
34 serv_sock = socket(PF_INET, SOCK_STREAM, 0);
35 memset(&serv_addr, 0, sizeof(serv_addr));
36 serv_addr.sin_family = AF_INET;
37 serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
38 serv_addr.sin_port = htons(atoi(argv[1]));
39
40 if(bind(serv_sock, (struct sockaddr*)&serv_addr,
41 sizeof(serv_addr)) == -1)
42 error_handling("Bind() error");
43 if(listen(serv_sock, 5) == -1)
44 error_handling("listen() error");
45 //创建epoll例程
46 epfd = epoll_create(E POLL_SIZE);
47 ep_events = malloc(sizeof(struct
48 epoll_event)*E POLL_SIZE); //分配内存
49
50 event.events = E POLLIN; //用来保存服务器发生变化的文件描
51述符
52 event.data.fd = serv_sock;
53 epoll_ctl(epfd, E POLL_CTL_ADD, serv_sock, &event);
54 //epoll例程epfd中注册文件描述符serv_sock, 主要目的是监视参数
55 event的事件E POLLIN
56
57 while(1)
58 {
59 event_cnt = epoll_wait(epfd, ep_events,
60 E POLL_SIZE, -1);
61 //成功时返回发生事件的文件描述符数, 失败时返回-1
62 //通过ep_events保存发生事件的文件描述符集合
63 if(event_cnt == -1)
64 {
65 puts("epoll_wait() error");
66 break;
67 }
68 for(i = 0; i < event_cnt; i++)
69 {
70 if(ep_events[i].data.fd == serv_sock) //检测
71 到新的连接请求
72 {
73 adr_sz = sizeof(cli nt_addr);
```

```
67 cInt_sock = accept(serv_sock, (struct
68 sockaddr*)&cInt_addr, &adr_sz);
69 event.events = EPOLLIN;
70 event.data.fd = cInt_sock;
71 epoll_ctl(epfd, EPOLL_CTL_ADD,
72 cInt_sock, &event);
73 printf("connected client: %d \n",
74 cInt_sock);
75 }
76 else //已经连接的其他client有数据需要发送，则read
77 并echo
78 {
79 str_len = read(ep_events[i].data.fd,
80 buf, BUF_SIZE);
81 if(str_len == 0) //收到连接断开的情
82 形,EPOLLRDHUP
83 {
84 epoll_ctl(epfd, EPOLL_CTL_DEL,
85 ep_events[i].data.fd, NULL);
86 close(ep_events[i].data.fd);
87 printf("closed client: %d \n",
88 ep_events[i].data.fd);
89 }
90 close(serv_sock);
91 close(epfd);
92 return 0;
93 }
94 void error_handling(char* message)
95 {
96 fputs(message, stderr);
97 fputc('\n', stderr);
98 exit(1);
```

## 17.4 条件触发(Level Trigger)和边缘触发(Edge Trigger)

两者区别在于发生事件的时间点

1. 条件触发方式中，只要输入缓冲有数据就会一直通知该事件
2. 边缘触发方式：在输入缓冲收到数据时仅注册1次该事件，即使输入缓冲中还留有数据，也不会再进行注册

epoll默认以条件触发方式工作

## 17.5 实现边缘触发的回声服务器端

```
57. event.events=EPOLLIN; event.events = EPOLLIN | EPOLLET;
```

**select模式是条件触发还是边缘触发：**

select模式是以**条件触发**的方式工作的，输入缓冲中如果还剩有数据，肯定会注册事件

### 17.5.1 边缘触发的服务器端实现中必知的两点

#### 1. 通过 `errno` 变量验证错误原因

linux声明了全局变量 `int errno`，为了访问该变量，需要引入 `error.h` 头文件。每种套接字相关函数发生错误时，保存到 `errno` 变量中的值都不同。

1. `read` 函数发现缓冲中没有数据可读时返回 -1，同时在 `errno` 中保存 `EAGAIN` 常量

#### 2. 为了完成非阻塞I/O，更改套接字特性

linux 提供更改或读取文件属性的方法 `fctl`

```

1 #include <fcntl.h>
2
3 int fcntl(int filedes, int cmd, ...); //成功时返回
4 cmd参数相关值，失败时返回-1
5 /*
6 filedes: 属性更改目标的文件描述符
7 cmd: 表明函数调用的目的
8 1. F_GETFL:可以获得第一个参数所指的文件描述符类型
9 (int)
10 2. F_SETFL:可以更改文件描述符属性
11 */
12 int flag = fcntl(fd, F_GETFL, 0); //获取之前设置的
13 属性信息
14 fcntl(fd, F_SETFL, flag|O_NONBLOCK); //将文件(套接
字)更改为非阻塞模式

```

边缘触发的服务器端实现与[读取错误原因](#)和[非阻塞模式的套接字创建](#)有着密切联系。

- 首先需要通过[errno](#)确认错误原因。因为在边缘触发方式中，接受数据时仅注册1次该事件。因此，需要验证输入缓冲是否为空。[read函数返回-1且变量errno中的值为EAGAIN时，说明没有数据可读](#)
- 在边缘触发方式下，阻塞方式工作的read和write函数有可能引起服务器端的长时间停顿。因此，边缘触发方式中一定要采用非阻塞的read和write函数

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <arpa/inet.h>
6 #include <sys/socket.h>
7 #include <sys/epoll.h>
8 #include <errno.h>
9 #include <fcntl.h>
10
11 #define BUF_SIZE 4 //为了验证边缘触发的工作方式，将缓冲设置为
12 4个字节
13 #define EPOLL_SIZE 50

```

```
13
14 void setnonblockingmode(int fd);
15 void error_handling(char* message);
16
17 int main(int argc, char* argv[])
18 {
19 int serv_sock, clnt_sock;
20 struct sockaddr_in serv_addr, clnt_addr;
21 socklen_t adr_sz;
22 int str_len, i;
23 char buf[BUF_SIZE];
24
25 struct epoll_event* ep_events;
26 struct epoll_event event;
27
28 int epfd, event_cnt; //epfd是通过epoll_create创建的
29 //epoll例程
30
31 if(argc!=2)
32 {
33 printf("Usage: %s <port> \n", argv[0]);
34 exit(1);
35 }
36
37 serv_sock = socket(PF_INET, SOCK_STREAM, 0);
38 memset(&serv_addr, 0, sizeof(serv_addr));
39 serv_addr.sin_family = AF_INET;
40 serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
41 serv_addr.sin_port = htons(atoi(argv[1]));
42
43 if(bind(serv_sock, (struct sockaddr*)&serv_addr,
44 sizeof(serv_addr)) == -1)
45 error_handling("Bind() error");
46 if(listen(serv_sock, 5) == -1)
47 error_handling("listen() error");
48
49 epfd = epoll_create(EPOLL_SIZE);
50 ep_events = malloc(sizeof(struct
```

```
51 event.events = EPOLLIN; //设置边缘触发
52 event.data.fd = serv_sock;
53 epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);
54
55 while(1)
56 {
57 event_cnt = epoll_wait(epfd, ep_events,
58 EPOLL_SIZE, -1); //通过epoll_wait等待判断文件描述符的事件变化
59 if(event_cnt == -1)
60 {
61 puts("epoll_wait() error");
62 break;
63 }
64 puts("return epoll_wait");
65 for(i = 0; i < event_cnt; i++)
66 {
67 if(ep_events[i].data.fd == serv_sock) //检测到新的连接请求
68 {
69 adr_sz = sizeof(cInt_addr);
70 cInt_sock = accept(serv_sock, (struct sockaddr*)&cInt_addr, &adr_sz);
71 setnonblockingmode(cInt_sock); //设置套接字的非阻塞模式
72 event.events = EPOLLIN|EPOLLET; //设置边缘触发
73 event.data.fd = cInt_sock;
74 epoll_ctl(epfd, EPOLL_CTL_ADD,
75 cInt_sock, &event);
76 printf("connected client: %d \n",
77 cInt_sock);
78 }
79 else
80 { while(1)
81 {//反复读取数据
82 str_len = read(ep_events[i].data.fd,
83 buf, BUF_SIZE);
84 if(str_len == 0)
85 {
```

```

83 epoll_ctl(epfd, EPOLL_CTL_DEL,
84 ep_events[i].data.fd, NULL);
85 close(ep_events[i].data.fd);
86 printf("closed client: %d \n",
87 ep_events[i].data.fd);
88 }
89 else if(str_len < 0)
90 {
91
92 if(errno == EAGAIN) break; //当前输入缓冲无数据
93 }
94 else
95 {
96 write(ep_events[i].data.fd, buf,
97 str_len); //echo!
98 }
99 }
100 close(serv_sock);
101 close(epfd);
102 return 0;
103 }
104 void setnonblockingmode(int fd)
105 {
106 int flag = fcntl(fd, F_GETFL, 0);
107 fcntl(fd, F_SETFL, flag|O_NONBLOCK);
108 }
109 void error_handling(char* message)
110 {
111 fputs(message, stderr);
112 fputc('\n', stderr);
113 exit(1);
114 }

```

## 17.6 问题探讨

### 1. 边缘触发vs条件触发

边缘触发的优点：可以分离接收数据和处理数据的时间点

条件触发也能够区分数据接收和处理，但在输入缓冲收到数据的情况下，如果不读取（延迟处理），则每次调用 `epoll_wait` 函数时都会产生响应的时间，而且事件数也会累加

## 2. 利用select函数实现服务器端时，代码层面存在的两个缺点是？

- 调用select函数后常见的针对所有文件描述符的循环语句
- 每次调用select函数时都需要向该函数传递监视对象信息

**优点：** epoll方式只在linux下提供支持，即改进的I/O复用模式不具有兼容性。而大多数操作系统都支持select函数。

## 3. 无论是select方式还是epoll方式，都需要将监视对象文件描述符信息通过函数调用传递给操作系统。请解释传递该信息的原因

“那为何需要把监视对象信息传递给操作系统呢？”

有些函数不需要操作系统的帮助就能完成功能，而有些则必须借助于操作系统。假设各位定义了四则运算相关函数，此时无需操作系统的帮助。但 `select` 函数与文件描述符有关，更准确地说，是监视套接字变化的函数。而套接字是由操作系统管理的，所以 `select` 函数绝对需要借助于操作系统才能完成功能。`select` 函数的这一缺点可以通过如下方式弥补：

“仅向操作系统传递1次监视对象，监视范围或内容发生变化时只通知发生变化的事项。”

这样就无需每次调用 `select` 函数时都向操作系统传递监视对象信息，但前提是操作系统支持这种处理方式（每种操作系统支持的程度和方式存在差异）。Linux的支持方式是 `epoll`，Windows 的支持方式是 `IOCP`。

即 `select` 函数与文件描述符有关，是监视套接字变化的函数。而 `套接字是由操作系统管理的`。所以 `select` 函数绝对需要借助于操作系统才能完成功能

## 4. select方式和epoll方式的最大差异在于监视对象文件描述符传递给操作系统的方式。请说明具体的差异，并解释为何存在这种差异。

epoll不同于 `select` 的地方是 `只要将监视对象文件描述符的信息传递一个给操作系统就可以了`。因此 `epoll` 方式克服了 `select` 方式的缺点，体现在 linux 内核上保存监视对象信息的方式。

## 5. 虽然 `epoll` 是 `select` 的改进方案，但是 `select` 也有自己的优点。在何种情况下使用 `select` 方式更合理？

如果连接服务器的人数不多（不需要高性能），而且需要在多种操作系统（windows和linux）下进行操作，在兼容性方面，使用 `select` 会比 `epoll` 更合理

## 6. `epoll` 以条件触发或边缘触发方式工作，二者有何区别？从输入缓冲的角度说明这两种方式通知事件的时间点差异

### 1. 条件触发方式中，只要输入缓冲有数据就会一直通知该事件

2. 边缘触发方式：在输入缓冲收到数据时仅注册1次该事件，即使输入缓冲中还留有数据，也不会再进行注册
7. 采用边缘触发时可以分离数据的接收和处理时间点，说明其原因和优点

如果使用边缘触发方式，在输入缓冲中接收数据时，只会发生一次事件通知，而且输入缓冲中仍有数据时，不会进行通知，因此可以在数据被接收后，在想要的时间内处理数据。而且，如果分离数据的接收和处理时间点，在服务器中会有更大的灵活性

## 18. 多线程服务器端的实现

**多进程模型存在的问题：**创建进程（复制）的工作本身会给操作系统带来相当沉重的负担。每个进程具有独立的内存空间，因此进程间通信难度也比较高。

- 创建进程的过程会带来一定的开销
- 为了完成进程间数据交换，需要特殊的IPC技术（管道通信）
- 每秒少则数十次，多则数千次的“上下文切换”是创建进程时最大的开销

**上下文切换：**为了进程能够实现分时使用CPU。运行程序前需要将相应进程信息读入内存，如果运行进程A后紧接着运行进程B，就应该将进程A相关信息移出内存，并读入进程B相关信息。

但此时进程A的数据将被移动到硬盘，所以上下文切换需要很长时间

为了保持多进程的优点，同时在一定程度上克服其缺点，人们引入了线程（Thread）。这是为了将进程的各种劣势降至最低限度（不是直接消除）而设计的一种“轻量级进程”。线程相比于进程具有如下优点。

- 线程的创建和上下文切换比进程的创建和上下文切换更快。
- 线程间交换数据时无需特殊技术。而进程间通信需要IPC技术

### 18.1 线程和进程的差异

每个进程的内存空间都由保存全局变量的“数据区”向malloc等函数的动态分配提供空间的堆（Heap）函数运行时使用的栈（Stack）构成。每个进程都拥有这种独立空间，多个进程的内存结构如图18-1所示。

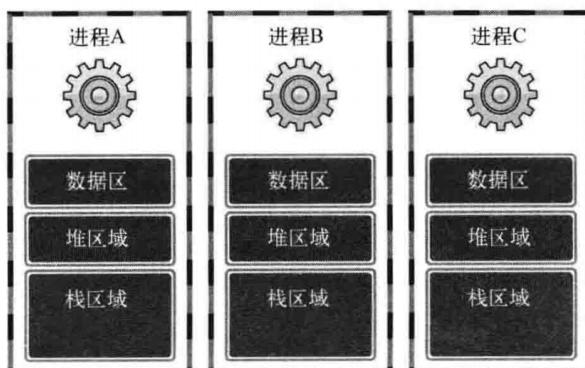


图18-1 进程间独立的内存

但如果以获得多个代码执行流为主要目的，则不应该像图18-1那样完全分离内存结构，而只需分离栈区域。通过这种方式可以获得如下优势。

- 上下文切换时不需要切换数据区和堆。
- 可以利用数据区和堆交换数据。

实际上这就是线程。线程为了保持多条代码执行流而隔开了栈区域，因此具有如图18-2所示的内存结构。

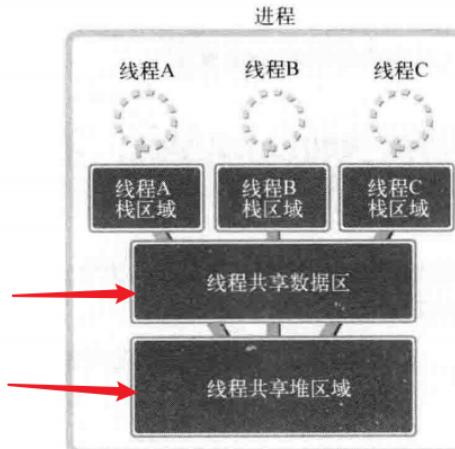


图18-2 线程的内存结构

如图18-2所示，多个线程将共享数据区和堆。为了保持这种结构，线程将在进程内创建并运行。也就是说，进程和线程可以定义为如下形式。  
★

- 进程：在操作系统构成单独执行流的单位。
- 线程：在进程中构成单独执行流的单位。

如果说进程在操作系统内部生成多个执行流，那么线程就在同一进程内部创建多条执行流。因此，操作系统、进程、线程之间的关系可以通过图18-3表示。

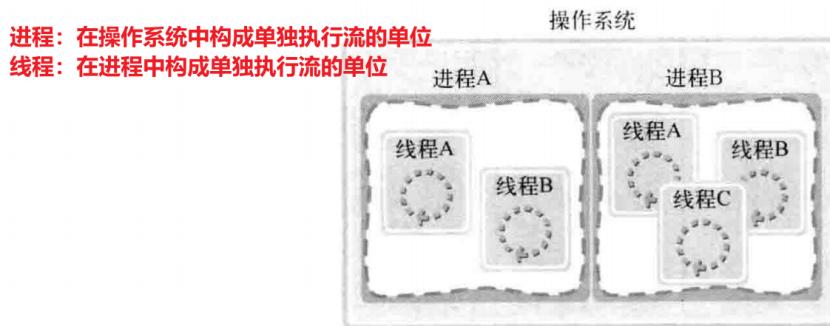


图18-3 操作系统、进程、线程之间的关系

## 18.2 线程创建及运行

Portable Operating System Interface for Computer Environment (**POSIX**) 是为了提高UNIX系列操作系统间的移植性而制定的API规范。

### 18.2.1 线程的创建和执行流程

线程具有单独的执行流，因此需要单独定义线程的main函数，还需要请求操作系统在单独的执行流中执行该函数。

```
1 #include <pthread.h>
2
3 int pthread_create(pthread_t* restrict thread, const
4 pthread_attr_t* restrict attr, void*(*start_routine)
5 (void*), void* restrict arg); //成功时返回0, 失败时返回其他值
6 /*
7 thread:保存新创建线程ID的变量地址值, 线程与进程相同, 也需要用于区
8 分不同线程的ID
9 attr:用于传递线程属性的参数, 传递NULL时, 创建默认属性的线程
10 start_routing:相当于线程main函数的, 在单独执行流中执行的函数地址
11 值(函数指针)
12 arg:通过第三个参数传递调用函数时包含传递参数信息的变量地址值
13 */
14
```

对于pthread\_create函数, 关键在于熟练掌握restrict关键字和函数指针相关语法

### thread1.c代码示例

```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 void* thread_main(void* arg);
5
6 int main(int argc, char* argv[])
7 {
8 pthread_t t_id;
9 int thread_param = 5;
10
11 if(pthread_create(&t_id, NULL, thread_main,
12 (void*)&thread_param) !=0)
13 {
14 //失败时返回其他值, 成功时返回0
15 puts("pthread_create() error");
16 return -1;
17 }
18
19 sleep(10); //调用sleep函数使main函数停顿10s, 为了延迟进程
20 //的终止时间。因为main函数在return以后, 会终止进程, 同时终止内部创
21 //建的进程。
22 puts("end of main");
23
24 return 0;
25 }
```

```

22 }
23
24 void* thread_main(void* arg) //传入参数为pthread_create函数的第四个参数
25 {
26 int i;
27 int cnt = *((int*)arg);
28 for(i = 0; i < cnt; i++)
29 {
30 sleep(1); puts("running thread");
31 }
32 return NULL;
33 }

```

```

[xiaotang@VM-0-5-centos srcPart2]$ gcc thread1.c -o thread1
/tmp/ccBxlv2.o: In function `main':
thread1.c:(.text+0x2f): undefined reference to `pthread_create'
collect2: error: ld returned 1 exit status
[xiaotang@VM-0-5-centos srcPart2]$ gcc thread1.c -o thread1 -lpthread
[xiaotang@VM-0-5-centos srcPart2]$./thread1
running thread
running thread
running thread
running thread
running thread
end of main
[xiaotang@VM-0-5-centos srcPart2]$

```

↑  
线程相关代码在编译时需要添加-lpthread  
选项声明需要连接线程库，只有这样才能调用头文件pthread.h中声明的函数

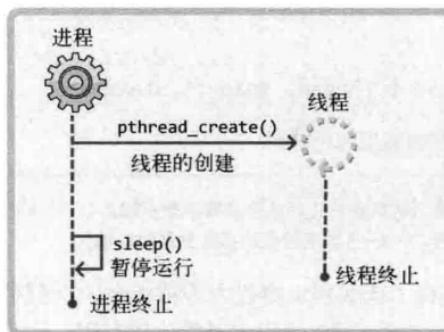


图18-4 示例thread1.c的执行流程

图18-4中的虚线代表执行流称，向下的箭头指的是执行流，横向箭头是函数调用。这些都是简单的符号，可以结合示例理解。接下来将上述示例的第15行sleep函数的调用语句改成如下形式：

```
sleep(2);
```

各位运行后可以看到，此时不会像代码中写的那样输出5次“running thread”字符串。因为main函数返回后整个进程将被销毁，如图18-5所示。

正因如此，我们在之前的示例中通过调用sleep函数向线程提供了充足的执行时间。

“那线程相关程序中必须适当调用sleep函数！”

并非如此！通过调用sleep函数控制线程的执行相当于预测程序的执行流程，但实际上这是不可能完成的事情。而且稍有不慎，很可能干扰程序的正常执行流。例如，怎么可能在上述示例中准确预测thread main函数的运行时间，并让main函数恰好等待这么长时间呢？因此，我们不用sleep函数，而是通常利用下面的函数控制线程的执行流。通过下列函数可以更有效地解决现讨论的问题，还可同时了解线程ID的用途。

## 18.2.2 通过pthread\_join函数控制线程的执行流

```
1 #include <pthread.h>
2
3 int pthread_join(pthread_t thread, void** status); //成功
时返回0，失败时返回其他值
4 /*
5 thread:该参数值ID的线程终止时才会从该函数返回
6 status:保存线程的main函数返回值的指针变量地址值
7 */
```

调用该函数的进程（或线程）将进入等待状态，直到第一个参数ID对应的线程终止为止，而且可以得到线程的main函数返回值。所以该函数比较有用

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <string.h>
5
6 void* thread_main(void* arg);
7
8 int main(int argc, char* argv[])
9 {
10 pthread_t t_id;
11 int thread_param = 5;
12
13 void* thr_ret; //thread的main函数返回值
14
15 if(pthread_create(&t_id, NULL, thread_main,
16 (void*)&thread_param) != 0)
17 {
18 puts("pthread_create() error");
19 return -1;
20 }
```

```
20
21 if(pthread_join(t_id, &thr_ret) != 0) //main函数将等
待ID保存在t_id变量中的线程终止
22 {
23 puts("pthread_join() error");
24 return -1;
25 }
26
27
28 printf("Thread return message: %s \n",
29 (char*)thr_ret);
30
31 free(thr_ret);
32 return 0;
33 }
34 void* thread_main(void* arg)
35 {
36 int i;
37 int cnt = *((int*)arg);
38 char* msg = (char*)malloc(sizeof(char)*50);
39
40 strcpy(msg, "hello, I am a thread!\n");
41
42 for(i = 0; i < cnt; ++i)
43 {
44 sleep(1); puts("running thread");
45 }
46
47 return (void*)msg; //返回线程内部动态分配的内存空间的地址
48 //容易内存泄露哈
49 }
50
51 int pthread_join(pthread_t thread, void** status); //成
功时返回0，失败时返回其他值
52 /*
53 thread:该参数值ID的线程终止时才会从该函数返回
54 status:保存线程的main函数返回值的指针变量地址值
55
56 线程返回值的获取方法:
```

```
57 1. 因为线程的主函数是void* thread_main(void* arg);因此通过
58 void* thr_ret来接受返回值，并将thr_ret传入pthread_join函数的
59 第二个参数
60 if(pthread_join(t_id, &thr_ret) != 0); //第二个参数为
61 void**
62 */
```

最后，为了让大家更好地理解该示例，给出其执行流程图，如图18-6所示。请注意观察程序暂停后从线程终止时（线程main函数返回时）重新执行的部分。

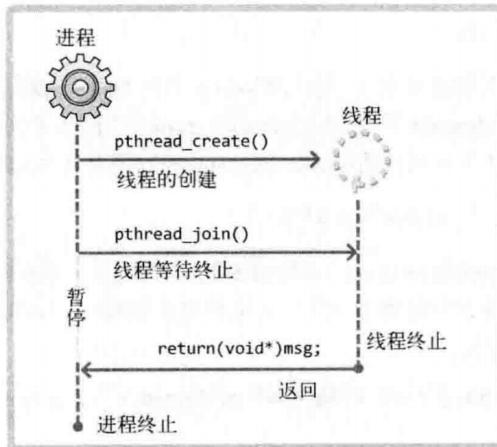


图18-6 调用pthread\_join函数

### 18.2.3 可在临界区内调用的函数

关于线程的运行需要考虑：多个线程同时调用函数时（执行时），这类函数内部存在临界区。即多个线程同时执行这部分代码时，可能引起问题，临界区中至少存在一条这类代码

根据临界区是否引起问题，函数可分为

#### 1. 线程安全函数 (Thread-safe function)

被多个线程同时调用时也不会引发问题，线程安全函数中同样可能存在临界区，只是在线程安全函数中，同时被多个线程调用时可通过一些措施避免问题

#### 2. 非~

幸运的是，大多数标准函数都是线程安全的函数。更幸运的是，我们不用自己区分线程安全的函数和非线程安全的函数（在Windows程序中同样如此）。因为这些平台在定义非线程安全函数的同时，提供了具有相同功能的线程安全的函数。比如，第8章介绍过的如下函数就不是线程安全的函数：

```
struct hostent * gethostbyname(const char * hostname); 非线程安全函数
```

同时提供线程安全的同一功能的函数。

```
struct hostent * gethostbyname_r(
 const char * name, struct hostent * result, char * buffer, intbuflen,
 int * h_errnop); 线程安全函数
```

线程安全函数的名称后缀通常为\_r（这与Windows平台不同）。既然如此，多个线程同时访问的代码块中应该调用gethostbyname\_r，而不是gethostbyname？当然！但这种方法会给程序员带来沉重的负担。幸好可以通过如下方法自动将gethostbyname函数调用改为gethostbyname\_r函数调用！

“声明头文件前定义\_\_REENTRANT宏。” **自动选用线程安全的标准函数**

gethostbyname函数和gethostbyname\_r函数的函数名和参数声明都不同，因此，这种宏声明方式拥有巨大的吸引力。另外，无需为了上述宏定义特意添加#define语句，可以在编译时通过添加  
**通过编译的时候添加-D\_REENTRANT选项进行宏定义**

```
root@my_linux:/tcpip# gcc -D_REENTRANT mythread.c -o mthread -lpthread
```

下面编译线程相关代码时均默认添加-D\_REENTRANT选项。

## 18.2.4 工作线程模型

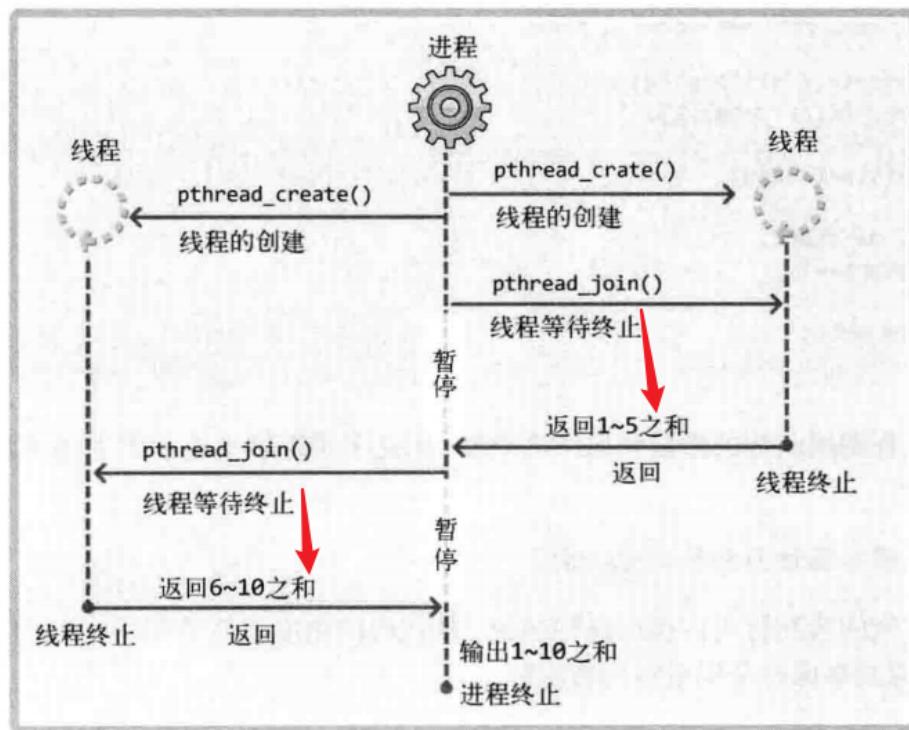


图18-7 示例thread3.c的执行流程

### thread3.c代码示例

```
1 #include <stdio.h>
2 #include <pthread.h>
3
```

```

4 void* thread_summation(void* arg);
5
6 int sum = 0; //全局变量区
7
8 int main(int argc, char* argv[])
9 {
10 pthread_t id_t1, id_t2;
11 int range1[] = {1, 5};
12 int range2[] = {6, 10};
13
14 pthread_create(&id_t1, NULL, thread_summation,
15 (void*)range1);
16 pthread_create(&id_t2, NULL, thread_summation,
17 (void*)range2);
18
19 pthread_join(id_t1, NULL);
20 pthread_join(id_t2, NULL);
21
22 printf("result: %d\n", sum);
23
24
25 void* thread_summation(void* arg)
26 {
27 int start = ((int*)arg)[0];
28 int end = ((int*)arg)[1];
29
30 while(start <= end)
31 {
32 sum+=start; //线程共享全局变量的数据区
33 start++;
34 }
35
36 return NULL;
37 }
```

## 具有更高发生临界区错误可能性的样例

```

for(i=0; i<NUM_THREAD; i++)
{
 if(i%2)
 pthread_create(&(thread_id[i]), NULL, thread_inc, NULL);
 else
 pthread_create(&(thread_id[i]), NULL, thread_des, NULL);
}

32. void * thread_inc(void * arg)
33. {
34. int i;
35. for(i=0; i<500000000; i++)
36. num+=1;
37. return NULL;
38. }
39. void * thread_des(void * arg)
40. {
41. int i;
42. for(i=0; i<500000000; i++)
43. num-=1;
44. return NULL;
45. }

```

创建100个线程，一半的线程进行thread\_inc  
另一半线程进行thread\_des

但是每个线程分别在做这个操作的时候，  
其他的线程可能还没有做完  
那么就会有多个线程同时访问全局变量num

## 18.3 线程存在的问题和临界区

### 18.3.1 多个线程访问同一变量

多线程编程中**同步**的重要性



图18-8 等待中的2个线程

### 18.3.2 临界区位置

**临界区定义：**函数内同时运行多个线程时引起问题的多条语句构成的代码块

全局变量num不是临界区，因为它不是引起问题的语句，只是代表内存区域的声明而已。

临界区通常位于线程运行的函数内部

```

1 | void* thread_inc(void* arg)
2 | {

```

```

3 int i;
4 for(i = 0; i < 50000000; i++)
5 num += 1; //临界区
6
7 return NULL;
8 }
9
10 void* thread_inc(void* arg)
11 {
12 int i;
13 for(i = 0; i < 50000000; i++)
14 num -= 1; //临界区
15
16 return NULL;
17 }
```

由代码注释可知，临界区并非num本身，而是访问num的2条语句。这2条语句可能由多个线程同时运行，也是引起问题的直接原因。产生的问题可以整理为如下3种情况。

- 2个线程同时执行thread\_inc函数。✓ 归根结底都是线程同时访问同一块内存空间
- 2个线程同时执行thread\_des函数。✓
- 2个线程分别执行thread\_inc函数和thread\_des函数。✓

需要关注最后一点，它意味着如下情况下也会引发问题：

“线程1执行thread\_inc函数的num+=1语句的同时，线程2执行thread\_des函数的num-=1语句。”

也就是说，2条不同语句由不同线程同时执行时，也有可能构成临界区。前提是这2条语句访问同一内存空间。



## 18.4 线程同步

用来解决线程访问顺序引发的问题

- 同时访问同一块内存空间时发生的情况（临界区）
- 需要指定访问同一内存空间的线程执行顺序的情况（控制线程执行顺序）

### 18.4.1 互斥量（锁机制）

互斥量是“Mutual Exclusion”的简写，表示不允许多个线程同时访问。主要用来解决线程同步访问的问题

#### 互斥量的创建及销毁函数

```

1 #include <pthread.h>
2
```

```

3 int pthread_mutex_init(pthread_mutex_t* mutex, const
4 pthread_mutexattr_t* attr);
5 /*
6 mutex: 创建互斥量时传递保存互斥量的变量地址值，销毁时传递需要销毁
7 的互斥量地址值
8 attr: 传递即将创建的互斥量属性，没有特别需要指定的属性时传递NULL
9 */
10 pthread_mutex_t mutex; //声明pthread_mutex_t型互斥量
11 //其地址值做为init和destroy的第一个参数
12
13 pthread_mutex_init(&mutex, NULL); //等价于
14 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
15 //但推荐使用pthread_mutex_init进行初始化，因为通过宏进行初始化
16 //时很难发现发生的错误

```

## 通过互斥量锁住或释放临界区时使用的函数

```

1 #include <pthread.h>
2
3 int pthread_mutex_lock(pthread_mutex_t* mutex);
4 int pthread_mutex_unlock(pthread_mutex_t* mutex); //成功
时返回0，失败时返回其他值

```

函数名本身含有lock、unlock等词汇，很容易理解其含义。进入临界区前调用的函数就是pthread\_mutex\_lock。调用该函数时，发现有其他线程已进入临界区，则pthread\_mutex\_lock函数不会返回，直到里面的线程调用pthread\_mutex\_unlock函数退出临界区为止。也就是说，其他线程让出临界区之前，当前线程将一直处于阻塞状态。接下来整理一下保护临界区的代码块编写方法。创建好互斥量的前提下，可以通过如下结构保护临界区。

```

pthread_mutex_lock(&mutex);
// 临界区的开始
// . . .
// 临界区的结束
pthread_mutex_unlock(&mutex);

```

简言之，就是利用lock和unlock函数围住临界区的两端。此时互斥量相当于一把锁，阻止多个线程同时访问。还有一点需要注意，线程退出临界区时，如果忘了调用pthread\_mutex\_unlock函数，那么其他为了进入临界区而调用pthread\_mutex\_lock函数的线程就无法摆脱阻塞状态。这

死锁

## 18.4.2 信号量

### 信号量创建及销毁方法

```
1 #include <semaphore.h>
2
3 int sem_init(sem_t* sem, int pshared, unsigned int
4 value);
5 int sem_destroy(sem_t* sem); //成功时返回0，失败时返回其他值
6 /*
7 sem: 创建信号量时传递保存信号量的变量地址值，销毁时传递需要销毁的信
8 号量变量
9 pshared: 传递其他值时，创建可由多个进程共享的信号量。传递0时，创建
10 只允许1个进程内部使用的信号量。
11 value: 指定新创建的信号量的初始值
12 */
13
14 //相当于互斥量lock和unlock的函数
15 int sem_post(sem_t* sem);
16 int sem_wait(sem_t* sem); //成功时返回0，失败时返回其他值
17 /*
18 *sem: 传递保存信号量读取值的变量地址值，传递给sem_post时信号量
19 *sem加1，传递给sem_wait时信号量减1
```

调用sem\_init函数时，操作系统将创建信号量对象，此对象中记录着“信号量值”(Semaphore Value)整数。该值在调用sem\_post函数时增1，调用sem\_wait函数时减1。但信号量的值不能小于0，因此，在信号量为0的情况下调用sem\_wait函数时，调用函数的线程将进入阻塞状态(因为函数未返回)。当然，此时如果有其他线程调用sem\_post函数，信号量的值将变为1，而原本阻塞的线程可以将该信号量重新减为0并跳出阻塞状态。实际上就是通过这种特性完成临界区的同步操作，可以通过如下形式同步临界区(假设信号量的初始值为1)。

```
sem_wait(&sem); // 信号量变为 0. . .
// 临界区的开始
. . .
// 临界区的结束
sem_post(&sem); // 信号量变为 1. . .
```

上述代码结构中，调用sem\_wait函数进入临界区的线程在调用sem\_post函数前不允许其他线程进入临界区。信号量的值在0和1之间跳转，因此，具有这种特性的机制称为“二进制信号量”。接下来给出信号量相关示例。即将介绍的示例并非关于同时访问的同步，而是关于控制访问顺序的同步。该示例的场景如下：

“线程A从用户输入得到值后存入全局变量num，此时线程B将取走该值并累加。该过程共进行5次，完成后输出总和并退出程序。”

### 代码实现：

```
1 #include <stdio.h>
```

```
2 #include <pthread.h>
3 #include <semaphore.h>
4
5 void* read(void* arg);
6 void* accu(void* arg);
7
8 static sem_t sem_one;
9 static sem_t sem_two; //两个信号量
10 static int num;
11 int main(int argc, char* argv[])
12 {
13 pthread_t id_t1, id_t2;
14 sem_init(&sem_one, 0, 0); //一个进程，初始值value=0
15 sem_init(&sem_two, 0, 1); //一个进程中，初始值value =
16 1
17 pthread_create(&id_t1, NULL, read, NULL);
18 pthread_create(&id_t2, NULL, accu, NULL);
19
20 pthread_join(id_t1, NULL);
21 pthread_join(id_t2, NULL);
22
23 sem_destroy(&sem_one);
24 sem_destroy(&sem_two);
25
26 return 0;
27 }
28
29 void* read(void* arg)
30 {
31 int i;
32 for(i = 0; i < 5; i++)
33 {
34 fputs("Input num: ", stdout);
35 sem_wait(&sem_two); //只有sem_two不为0的时候才能够执
行scanf
36 scanf("%d", &num);
37 sem_post(&sem_one);
38 }
39 return NULL;
40 }
```

```

41
42 void* accu(void* arg)
43 {
44 int sum = 0, i;
45 for(i = 0; i<5; i++)
46 {
47 sem_wait(&sem_one);
48 sum+=num;
49 sem_post(&sem_two); //这时候sem_two变为1
50 }
51 printf("Result: %d \n", sum);
52
53 return NULL;
54 }
```

**利用两个信号量sem\_one和sem\_two，并通过调用sem\_wait和sem\_post函数，实现0,1的翻转**

- 第35、48行：利用信号量变量sem\_two调用wait函数和post函数。这是为了防止在调用accu函数的线程还未取走数据的情况下，调用read函数的线程覆盖原值。
- 第37、46行：利用信号量变量sem\_one调用wait和post函数。这是为了防止调用read函数的线程写入新值前，accu函数取走（再取走旧值）数据。

## 18.5 线程的销毁和多线程并发服务器端的实现

### 18.5.1 销毁线程的3种方法

Linux线程并不是在首次调用的线程main函数返回时自动销毁，所以需要通过如下两种方法之一加以明确，否则由线程创建的内存空间将一直存在

#### 1. 调用pthread\_join函数

调用该函数不仅会等待线程终止，还会引导线程销毁。

**存在的问题：**调用该函数的线程将进入阻塞状态，因此通常通过  
`pthread_detach`函数引导线程销毁

#### 2. 调用pthread\_detach函数

```

1 #include <pthread.h>
2
3 int pthread_detach(pthread_t thread); //成功时返回0，失败时返回其他值
4 //thread: 终止的同时需要销毁的线程ID
```

调用该函数不会引起线程终止或进入阻塞状态，可以通过该函数引导销毁线程创建的内存空间。

调用该函数后不能再针对相应线程调用 `pthread_join` 函数

## 18.5.2 多线程并发服务器端的实现

实现功能，涉及知识点：

1. 线程的使用方法

- 创建、同步、销毁

2. 临界区的处理方式

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/socket.h>
4 #include <string.h>
5 #include <arpa/inet.h>
6 #include <pthread.h>
7 #include <semaphore.h>
8
9 #define BUF_SIZE 100
10#define MAX_CLNT 256
11
12void* handle_clnt(void* arg);
13void send_msg(char* msg, int len);
14void error_handling(char* msg);
15
16int clnt_cnt = 0;
17int clnt_socks[MAX_CLNT]; //管理接入的客户端套接字的变量和数组，访问这两个变量的代码将构成临界区
18pthread_mutex_t mutex;
19
20int main(int argc, char* argv[])
21{
22 int serv_sock, clnt_sock;
23 struct sockaddr_in serv_addr, clnt_addr;
24 int clnt_addr_sz;
25 pthread_t t_id;
26
27 if(argc != 2)
28 {
29 printf("Usage : %s <port> \n", argv[0]);
```

```
30 exit(1);
31 }
32
33 pthread_mutex_init(&mutex, NULL);
34 serv_sock = socket(PF_INET, SOCK_STREAM, 0);
35
36 memset(&serv_addr, 0, sizeof(serv_addr));
37 serv_addr.sin_family = AF_INET; //IPv4地址
38 serv_addr.sin_port = htons(atoi(argv[1]));
39 serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
40 if(bind(serv_sock, (struct sockaddr*)&serv_addr,
41 sizeof(serv_addr)) == -1)
42 error_handling("bind() error");
43 if(listen(serv_sock, 5) == -1)
44 error_handling("listen() error");
45
46 while(1)
47 {
48 cInt_adr_sz = sizeof(cInt_adr);
49 cInt_sock = accept(serv_sock, (struct
50 sockaddr*)&cInt_adr, &cInt_adr_sz);
51
52 pthread_mutex_lock(&mutex);
53 cInt_socks[cInt_cnt++] = cInt_sock; //临界区，因
54 为涉及到操作全局变量
55 pthread_mutex_unlock(&mutex);
56
57 pthread_create(&t_id, NULL, handle_cInt,
58 (void*)&cInt_sock); //创建线程向新接入的客户端提供服务，并执
59 行handle_cInt函数，并将cInt_sock作为参数传入
60 pthread_detach(t_id); //通过pthread_detach销毁线
61 程
62 printf("Connected client IP: %s \n",
63 inet_ntoa(cInt_adr.sin_addr)); //将网络序转为字节序?
64 }
65
66 void* handle_cInt(void* arg)
67 {
```

```
64 int clnt_sock = *((int*)arg);
65 int str_len = 0, i;
66 char msg[BUF_SIZE];
67
68 while((str_len = read(clnt_sock, msg,
69 sizeof(msg)))!=0)
70 send_msg(msg, str_len);
71
72 pthread_mutex_lock(&mutex);
73 for(i = 0; i < clnt_cnt; i++) //remove
74 disconnected client
75 {
76 if(clnt_sock == clnt_socks[i])
77 {
78 while(i++ < clnt_cnt-1)
79 clnt_socks[i] = clnt_socks[i+1];
80 break;
81 }
82 clnt_cnt--;
83 pthread_mutex_unlock(&mutex);
84 close(clnt_sock);
85
86 return NULL;
87 }
88 void send_msg(char* msg, int len) //send to all
89 {
90 int i;
91 pthread_mutex_lock(&mutex);
92 for(i = 0; i < clnt_cnt; i++)
93 {
94 write(clnt_socks[i], msg, len);
95 }
96 pthread_mutex_unlock(&mutex);
97 }
98
99 void error_handling(char* message)
100 {
101 fputs(message, stderr);
102 fputc('\n', stderr);
```

```
103 exit(1);
104 }
```

上述示例中，各位必须掌握的并不是聊天服务器端的实现方式，而是临界区的构成形式。上述示例中的临界区具有如下特点：

“访问全局变量clnt\_cnt和数组clnt\_socks的代码将构成临界区！”

添加或删除客户端时，变量clnt\_cnt和数组clnt\_socks同时发生变化。因此，在如下情形中均会导致数据不一致，从而引发严重错误。

- 线程A从数组clnt\_socks中删除套接字信息，同时线程B读取clnt\_cnt变量。
- 线程A读取变量clnt\_cnt，同时线程B将套接字信息添加到clnt\_socks数组。

因此，如上述示例所示，访问变量clnt\_cnt和数组clnt\_socks的代码应组织在一起并构成临界区。大家现在应该对我之前说过的这句话有同感了吧：

```
#include <sys/socket.h>

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

→ 成功时返回创建的套接字文件描述符，失败时返回-1。

- sock 服务器套接字的文件描述符。
- addr 保存发起连接请求的客户端地址信息的变量地址值，调用函数后向传递来的地址变量填充客户端地址信息。
- addrlen 第二个参数addr结构体的长度，但是存有长度的变量地址。函数调用完成后，该变量即被填入客户端地址长度。

accept函数受理连接请求等待队列中待处理的客户端连接请求。函数调用成功时，accept函数内部将产生用于数据I/O的套接字，并返回其文件描述符。需要强调的是，套接字是自动创建的，并自动与发起连接请求的客户端建立连接。图4-8展示了accept函数调用过程。

服务器端

### 18.5.3 多线程并发客户端实现

#### 实现输入输出分离

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/socket.h>
4 #include <arpa/inet.h>
5 #include <pthread.h>
6 #include <string.h>
7
8 #define BUF_SIZE 100
9 #define NAME_SIZE 20
10
11 void* send_msg(void* arg);
12 void* recv_msg(void* arg); // 定义收发信息的函数，交由线程处理
13 void error_handling(char* message);
14
```

```
15 char name[NAME_SIZE] = "[DEFAULT]";
16 char msg[BUF_SIZE];
17
18 int main(int argc, char* argv[])
19 {
20 int sock;
21 struct sockaddr_in serv_addr;
22 pthread_t snd_thread, rcv_thread; //收发数据的线程
23 void* thread_return;
24
25 if(argc != 4)
26 {
27 printf("Usage: %s <IP> <port> <name> \n",
28 argv[0]);
29 exit(1);
30 }
31
32 sprintf(name, "[%s]", argv[3]); //将argv[3]以[%s]的形式输出到name保存
33
34 sock = socket(PF_INET, SOCK_STREAM, 0);
35
36 memset(&serv_addr, 0, sizeof(serv_addr));
37 serv_addr.sin_family = AF_INET;
38 serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
39 serv_addr.sin_port = htons(atoi(argv[2]));
40
41 if(connect(sock, (struct sockaddr*)&serv_addr,
42 sizeof(serv_addr)) == -1)
43 error_handling("connect() error");
44
45 pthread_create(&snd_thread, NULL, send_msg,
46 (void*)&sock);
47 pthread_create(&rcv_thread, NULL, recv_msg,
48 (void*)&sock);
49
50 pthread_join(snd_thread, &thread_return);
51 pthread_join(rcv_thread, &thread_return);
52
53 close(sock);
54
55 return 0;
```

```
51 }
52
53 void* send_msg(void* arg)
54 {
55 int sock = *((int*)arg);
56 char name_msg[NAME_SIZE + BUF_SIZE];
57
58 while(1)
59 {
60 fgets(msg, BUF_SIZE, stdin);
61 if(!strcmp(msg, "q\n") || !strcmp(msg, "Q\n"))
62 {
63 close(sock);
64 exit(0);
65 }
66
67 sprintf(name_msg, "%s %s", name, msg);
68 write(sock, name_msg, strlen(name_msg));
69 }
70
71 return NULL;
72 }
73
74 void* recv_msg(void* arg) //read thread_main
75 {
76 int sock = *((int*)arg);
77 char name_msg[NAME_SIZE + BUF_SIZE];
78 int str_len;
79
80 while(1)
81 {
82 str_len = read(sock, name_msg, NAME_SIZE +
83 BUF_SIZE - 1);
84 if(str_len == -1)
85 return (void*)-1;
86
87 name_msg[str_len] = 0;
88 fputs(name_msg, stdout);
89 }
90
91 return NULL;

```

```
91 }
92
93 void error_handling(char* msg)
94 {
95 fputs(msg, stderr);
96 fputc('\n', stderr);
97 exit(1);
98 }
```

## 18.6 问题探讨

1. 单CPU系统中如何同时执行多个进程？请解释该过程中发生的上下文切换

因为系统将CPU切分成多个微小的块后分配给多个进程，为了分时使用CPU，需要“上下文切换”过程。

“上下文切换”是指，在CPU改变运行对象的过程中，执行准备的过程将之前执行的进程数据从换出内存，并将待执行额进程数据传到内存的工作区域

2. 为何线程的上下文切换速度相对更快？线程间数据交换为何不需要类似IPC的特别技术？

因为线程进行上下文切换时不需要切换数据区和堆区。同时，可以利用数据区和堆区进行数据交换

3. 请从执行流角度说明进程和线程的区别

- 进程：在操作系统中构成单独执行流的单位
- 线程：在进程内构成单独执行流的单位

(4) 下列关于临界区的说法错误的是？

- a. 临界区是多个线程同时访问时发生问题的区域。
- b. 线程安全的函数中不存在临界区，即便多个线程同时调用也不会发生问题。 ×
- c. 1个临界区只能由1个代码块，而非多个代码块构成。换言之，线程A执行的代码块A和线程B执行的代码块B之间绝对不会构成临界区。
- d. 临界区由访问全局变量的代码构成。其他变量中不会发生问题。

(5) 下列关于线程同步的描述错误的是？

- a. 线程同步就是限制访问临界区。
- b. 线程同步也具有控制线程执行顺序的含义。
- c. 互斥量和信号量是典型的同步技术。
- d. 线程同步是代替进程IPC的技术。 IPC: 解决进程间通信问题 ×

6. 请说明完全销毁linux线程的2种方法

- `pthread_join` 函数和 `pthread_detach` 函数

7. 利用多线程技术实现回声服务器端，并让所有线程共享保存客户端消息的内存空间（char数组）

```
1 //echo_thrserv.c
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <unistd.h>
6 #include <arpa/inet.h>
7 #include <sys/socket.h>
8 #include <sys/select.h>
9 #include <pthread.h>
10
11 #define BUF_SIZE 100
12 void * handle_clnt(void * arg);
13 void error_handling(char *buf);
14
15 char buf[BUF_SIZE];
16 pthread_mutex_t mutx;
17
18 int main(int argc, char *argv[])
19 {
20 int serv_sock, clnt_sock;
21 struct sockaddr_in serv_addr, clnt_addr;
22 int clnt_adr_sz;
23 pthread_t t_id;
24
25 if(argc!=2) {
26 printf("Usage : %s <port>\n", argv[0]);
27 exit(1);
28 }
29
30 pthread_mutex_init(&mutx, NULL);
31 serv_sock=socket(PF_INET, SOCK_STREAM, 0);
32 memset(&serv_addr, 0, sizeof(serv_addr));
33 serv_addr.sin_family=AF_INET;
34 serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
35 serv_addr.sin_port=htons(atoi(argv[1]));
36
```

```
37 if(bind(serv_sock, (struct sockaddr*) &serv_addr,
38 sizeof(serv_addr)) == -1)
39 error_handling("bind() error");
40 if(listen(serv_sock, 5) == -1)
41 error_handling("listen() error");
42
43 while(1)
44 {
45 cInt_adr_sz = sizeof(cInt_adr);
46 cInt_sock = accept(serv_sock, (struct
47 sockaddr*)&cInt_adr, &cInt_adr_sz);
48 pthread_create(&t_id, NULL, handle_cInt,
49 (void*)&cInt_sock);
50 pthread_detach(t_id);
51 printf("Connected client IP: %s \n",
52 inet_ntoa(cInt_adr.sin_addr));
53 }
54
55 close(serv_sock);
56 return 0;
57 }
58
59 void * handle_cInt(void * arg)
60 {
61 int cInt_sock = *((int*)arg);
62 int str_len = 0;
63
64 while(1)
65 {
66 pthread_mutex_lock(&mutex);
67 str_len = read(cInt_sock, buf, sizeof(buf));
68 if(str_len <= 0)
69 break;
70 else
71 write(cInt_sock, buf, str_len);
72 pthread_mutex_unlock(&mutex);
73 }
74
75 close(cInt_sock);
76 return NULL;
77 }
```

```
74 void error_handling(char *buf)
75 {
76 fputs(buf, stderr);
77 fputc('\n', stderr);
78 exit(1);
79 }
```

存在的问题：

如果不进行线程同步，两个以上线程会同时访问内存空间，从而引发问题。

相反，如果同步，由于read函数中调用了临界区的参数，可能会发生无法读取其他客户端发送过来的字符串而必须等待的情况

## 19. 总结

### 19.1 关于int main中的传入参数

```
1 int main(int argc, char** argv)
2 {
3 //函数体内使用或不使用argc和argv都可
4
5 return 0;
6 }
```

#### 一、argc、argv的具体含义

##### 一、argc、argv的具体含义

argc和argv参数在用命令行编译程序时有用。main( int argc, char\* argv[], char \*\*env ) 中

第一个参数，int型的argc，为整型，用来统计程序运行时发送给main函数的命令行参数的个数，在VS中默认值为1。

第二个参数，char\*型的argv[]，为字符串数组，用来存放指向的字符串参数的指针数组，每一个元素指向一个参数。各成员含义如下：

argv[0]指向程序运行的全路径名

argv[1]指向在DOS命令行中执行程序名后的第一个字符串

argv[2]指向执行程序名后的第二个字符串

argv[3]指向执行程序名后的第三个字符串

argv[argc]为NULL

```
1 #include <stdio.h>
2 using namespace std;
3
4 int main(int argc, char ** argv)
5 {
6 int i;
7 printf("Argc = %d", argc);
8 for (i = 0; i < argc; i++)
9 printf("Argument %d is %s\n", i, argv[i]);
10 return 0;
11 }
```

```
[xiaotang@VM-0-5-centos src]$./testMain a b c d
Argc = 5
Argument 0 is ./testMain
Argument 1 is a
Argument 2 is b
Argument 3 is c
Argument 4 is d
```

因为命令行中argument有五个，因此argc = 5

## 19.2 socket的read和write

### 19.2.1 write函数

一旦，我们建立好了tcp连接之后，我们就可以把得到的fd当作文件描述符来使用。

ssize\_t **write**(int fd, const void\*buf,size\_t nbytes);

**write**函数将buf中的nbytes字节内容写入文件描述符fd.成功时返回写的字节数.失败时返回-1. 并设置errno变量. 在网络程序中,当我们向套接字文件描述符写时有两可能.  
1)**write**的返回值大于0,表示写了部分或者是全部的数据. 这样我们用一个while循环来不停的写入，但是循环过程中的buf参数和nbyte参数得由我们来更新。也就是说，网络写函数是不负责将全部数据写完之后在返回的。

2)返回的值小于0,此时出现了错误.我们要根据错误类型来处理.

如果错误为EINTR表示在写的时候出现了中断错误.

如果为EPIPE表示网络连接出现了问题(对方已经关闭了连接).

#### 从**write**函数返回的时间点:

**write**函数和windows的**send**函数 **并不会在完成向对方主机的数据传输时返回，而是在数据移到输出缓冲时。** 但TCP会保证对输出缓冲数据的传输，所以**write**函数在数据传输完成时返回。

## 19.2.2 读函数read

```
ssize_t read(int fd,void *buf,size_t nbyte)
```

read函数是负责从fd中读取内容.当读成功时,read返回实际所读的字节数,如果返回的值是0表示已经读到文件的结束了,小于0表示出现了错误.如果错误为EINTR说明读是由中断引起的,如果是ECONNRESET表示网络连接出了问题.

## 19.2.3 fread()和fwrite()

```
1 FILE* fp;
2 char buf[BUF_SIZE];
3 fp = fopen("fileName.type", "rb");
4
5 read_cnt = fread((void *)buf, 1, BUF_SIZE, fp);
6 fwrite((void*)buf, 1, read_cnt, fp);
7
8 fclose(fp);
```

## 19.2.4 readv和writev函数

**适用条件:** 如果要传输的数据分别位于不同的缓冲(数组)时,需要调用多次**write**函数。此时可以通过一次**writev**函数来替代。

从C语言的角度来看,减少函数调用次数能够相应提高性能。其更大的意义在于能够减少数据包的个数。假设为了提高效率而在服务器端明确阻止了Nagle算法。那么**writev**函数在不采用Nagle算法时更有价值

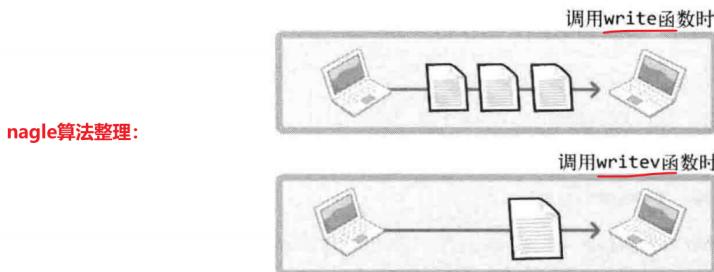


图13-5 Nagle算法关闭状态下的数据传输

上述示例中待发送的数据分别存在3个不同的地方,此时如果使用**write**函数则需要3次函数调用。但若为提高速度而关闭了Nagle算法,则极有可能通过3个数据包传递数据。反之,若使用**writev**函数将所有数据一次性写入输出缓冲,则很有可能仅通过1个数据包传输数据。所以**writev**函数和**readv**函数非常有用。

再考虑一种情况:将不同位置的数据按照发送顺序移动(复制)到1个大数组,并通过1次**write**函数调用进行传输。这种方式是否与调用**writev**函数的效果相同?当然!但使用**writev**函数更为便利。因此,如果遇到**writev**函数和**readv**函数的适用情况,希望各位不要错过机会。

## 19.3 TCP三次握手

套接字是以全双工（Full-duplex）方式工作的。也就是说，它可以双向传递数据。因此，收发数据前需要做一些准备。首先，请求连接的主机A向主机B传递如下信息：

[SYN] SEQ: 1000, ACK: -

该消息中SEQ为1000，ACK为空，而SEQ为1000的含义如下：

“现传递的数据包序号为1000，如果接收无误，请通知我向您传递1001号数据包。”

这是首次请求连接时使用的消息，又称SYN。SYN是Synchronization的简写，表示收发数据前传输的同步消息。接下来主机B向A传递如下消息：

[SYN+ACK] SEQ: 2000, ACK: 1001

此时SEQ为2000，ACK为1001，而SEQ为2000的含义如下：

“现传递的数据包序号为2000，如果接收无误，请通知我向您传递2001号数据包。”

而ACK 1001的含义如下：

“刚才传输的SEQ为1000的数据包接收无误，现在请传递SEQ为1001的数据包。”

对主机A首次传输的数据包的确认消息（ACK 1001）和为主机B传输数据做准备的同步消息（SEQ 2000）捆绑发送，因此，此种类型的消息又称SYN+ACK。

收发数据前向数据包分配序号，并向对方通报此序号，这都是为防止数据丢失所做的准备。通过向数据包分配序号并确认，可以在数据丢失时马上查看并重传丢失的数据包。因此，TCP可

以保证可靠的数据传输。最后观察主机A向主机B传输的消息：

[ACK] SEQ: 1001, ACK: 2001

之前也讨论过，TCP连接过程中发送数据包时需分配序号。在之前的序号1000的基础上加1，也就是分配1001。此时该数据包传递如下消息：

“已正确收到传输的SEQ为2000的数据包，现在可以传输SEQ为2001的数据包。”

这样就传输了添加ACK 2001的ACK消息。至此，主机A和主机B确认了彼此均就绪。

### 19.3.1 ACK增量

ACK号的增量为传输的数据字节数，假设每次ACK号不加传输的字节数，这样虽然可以确认数据包的传输，但是无法确定是否全部字节都正确传递，还是存在丢失情况

因此：ACK号 = SEQ号 + 传递的字节数 + 1；

最后加1是为了告知对方下次要传递的SEQ号

## 19.4 TCP四次挥手：断开套接字的连接

TCP套接字的结束过程也非常优雅。如果对方还有数据需要传输时直接断掉连接会出问题，所以断开连接时需要双方协商。断开连接时双方对话如下。

- 套接字A：“我希望断开连接。” **FIN**
  - 套接字B：“哦，是吗？请稍候。” **ACK**
- 中途可能还有些数据没有发送完毕
- 套接字B：“我也准备就绪，可以断开连接。” **FIN**
  - 套接字A：“好的，谢谢合作。” **ACK**

## 19.5 函数指针

<https://www.runoob.com/cprogramming/c-fun-pointer-callback.html>

## 19.6 sigaction的信号

- SIGALRM：已到通过调用alarm函数注册的时间
- SIGINT：输入ctrl + c
- SIGCHLD：子进程终止
- SIGURG：收到**MSG\_OOB**紧急消息

## 19.7 Nagle算法

纳格算法是以减少数据包发送量来增进TCP/IP网络的性能。

纳格的文件[\[注1\]](#)描述了他所谓的“小数据包问题” - 某个应用程序不断地提交小单位的资料，且某些常只占1字节大小。因为TCP数据包具有40字节的标头信息（TCP与IPv4各占20字节），这导致了41字节大小的数据包只有1字节的可用信息，造成庞大的浪费。这种状况常常发生于[Telnet](#)工作阶段 - 大部分的键盘操作会产生1字节的资料并马上提交。更糟的是，在慢速的网络连线下，这类的数据包会大量地在同一时间点传输，造成[壅塞碰撞](#)。

纳格算法的工作方式是合并（[coalescing](#)）一定数量的输出资料后一次提交。特别的是，只要有已提交的数据包尚未确认，发送者会持续缓冲数据包，直到累积一定数量的资料才提交。

## 19.8 epoll的本质

## 19.9 课后参考

[https://blog.csdn.net/nail\\_candy/article/details/88957409](https://blog.csdn.net/nail_candy/article/details/88957409)

[https://blog.csdn.net/nail\\_candy/article/details/88957438](https://blog.csdn.net/nail_candy/article/details/88957438)

