

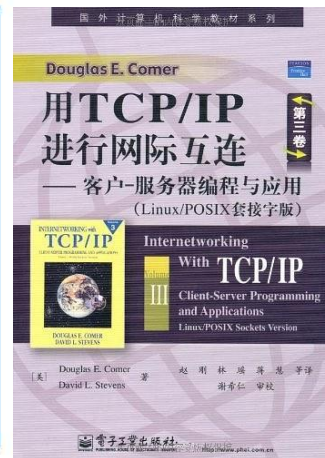
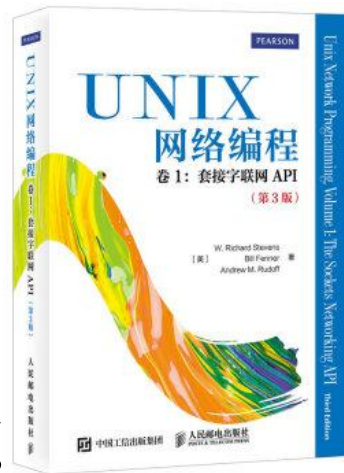
# 网络编程（4学时）

## □ 学习目标

- 掌握字节序、数据对齐等计算机网络编程相关基础知识
- 理解Socket基本概念以及TCP/UDP Socket编程基本模式
- 理解客户端、服务器程序设计的核心问题与解决思路
- 能够按需设计实现**简单循环服务器、多进程并发服务器**

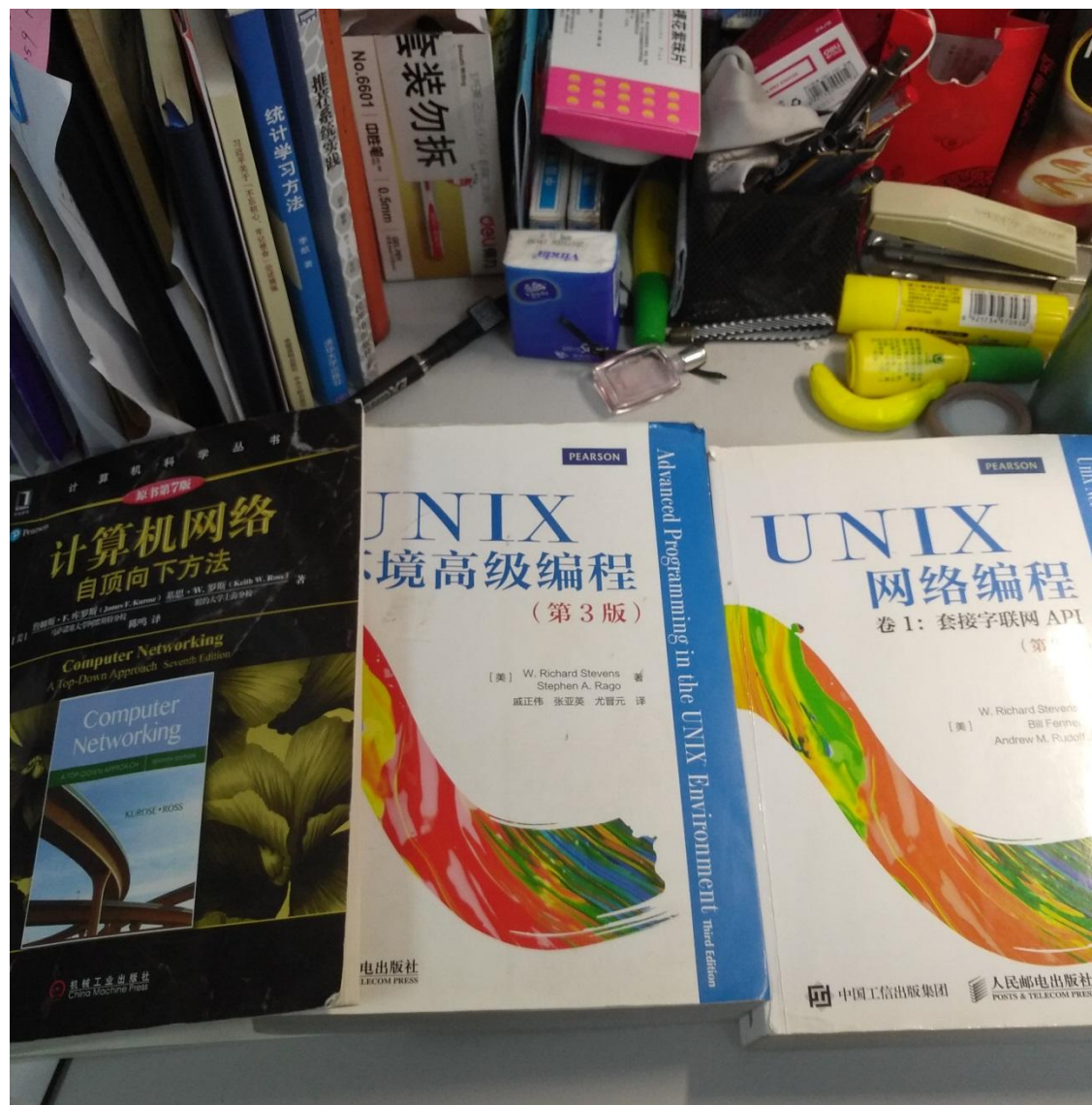
## □ 参考教材（非必须）

- UNIX网络编程（卷1：套接字联网API）（第三版），人民邮电出版社，2010
- 用TCP/IP进行网际互联(第3卷)：客户-服务器编程与应用(Linux版)，电子工业出版社，2004



# 内容安排

- ❑ 7.1 网络编程基础
- ❑ 7.2 套接字基础
- ❑ 7.3 循环服务器/客户端
- ❑ 7.4 并发服务器
- ❑ 7.5 代码优化-实验相关



## 7.1 网络编程基础

- ❑ 环境：Linux
- ❑ 编译GCC
- ❑ 常用Linux命令介绍
  - 显示当前目录：pwd
  - 转换目录：cd
  - 创建目录：mkdir
  - 删除文件与目录：rm
  - 修改文件的权限：chmod
  - 拷贝文件：cp
  - 显示文件列表：ls 或 ll
  - 清屏：clear
  - 执行程序：./程序名

## 7.1.1 网络编程环境搭建 (不考)

### □ 建立可执行程序步骤

- 建立源文件
- 建立目标文件
- 建立可执行文件

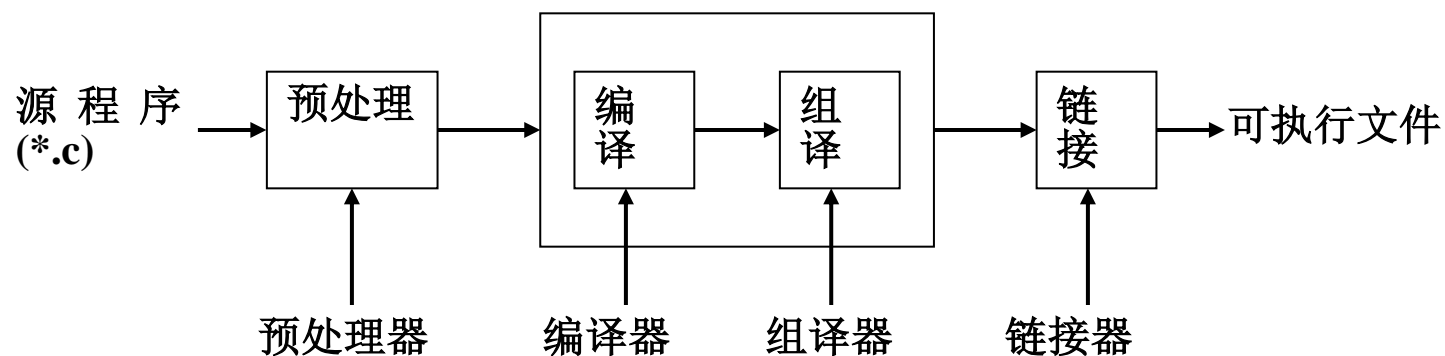
库文件指的是为用户程序和操作系统之间提供接口的程序

### □ 源代码

- 目标代码：由编译程序和解释程序把源代码翻译成机器能够理解的语言。目标代码不是可执行文件，它还缺少库文件。
- 可执行代码：可能包含其它程序代码，由链接程序将目标代码和其它程序代码链接在一起，形成完整的可执行程序

## 7.1.1 网络编程环境搭建-GCC编译工具(不考)

- GCC(GNU C Compiler)编译器：GNU推出的功能强大、性能优越的多平台编译器，gcc编译器能将C、C++语言源程序、汇编语言和目标程序编译、链接成可执行文件，以下是gcc支持编译的一些源文件的后缀及其解释（[点击](#)）
- 使用gcc将C源代码文件生成可执行文件，需要经历4个相关的步骤：预处理，编译，汇编，链接



# 网络编程环境搭建-GCC编译工具(不考)

## □ GCC的基本用法: gcc [选项] [文件名列表]

这里的文件名列表指的是需要**编译和链接**的文件，例如C, C++源文件，目标文件，汇编程序代码等。gcc的选项非常多，可以使用man gcc来查看，我们经常使用的选项是 -c -g -o

## □ GCC的选项说明:

- -c 只编译并生成目标文件(.o)
- -g 生成调试信息
- -o file 指定生成的文件名



# 网络编程环境搭建-GCC编译示例(不考)

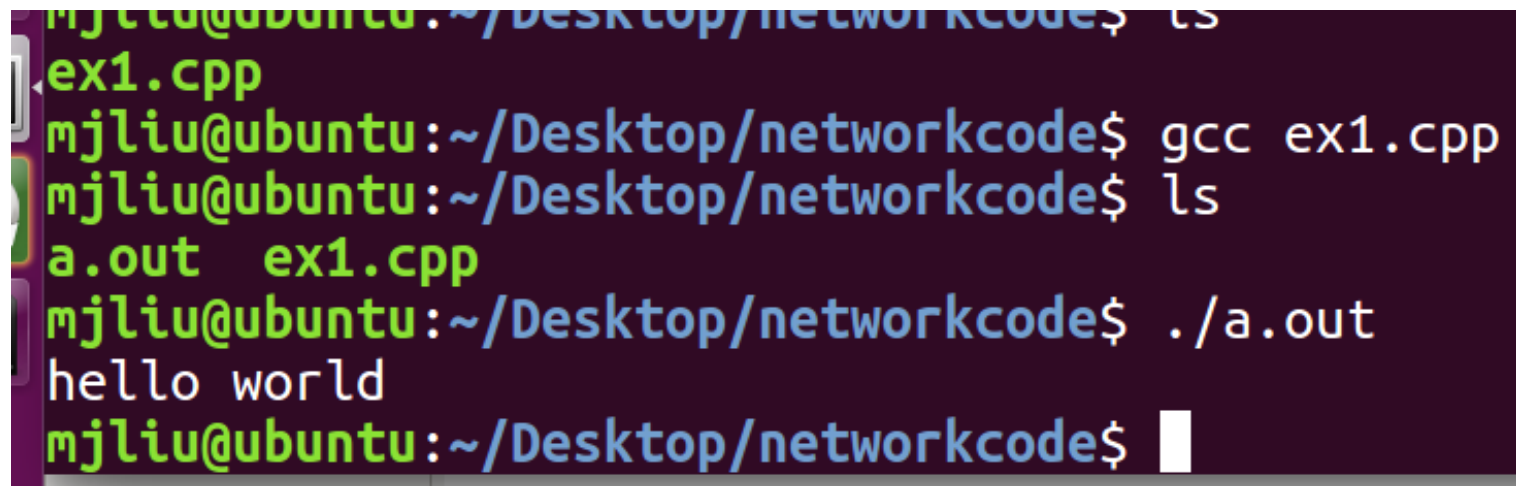
`gcc test.c`                                `-> a.out`    (不指定文件名，默认生成a.out)

`gcc -c test.c`                            `-> test.o`

`gcc -o test test.c`                      `-> test`

`gcc -o test test1.o test2.o` `-> test`

`gcc -o test test.o -lmath` `-> test`



```
mjliu@ubuntu:~/Desktop/networkcode$ ls
ex1.cpp
mjliu@ubuntu:~/Desktop/networkcode$ gcc ex1.cpp
mjliu@ubuntu:~/Desktop/networkcode$ ls
a.out  ex1.cpp
mjliu@ubuntu:~/Desktop/networkcode$ ./a.out
hello world
mjliu@ubuntu:~/Desktop/networkcode$
```

## 7.1.2 网络程序C/S架构

### □ 计算机网络的基础知识

- 网络应用程序的架构: C/S
- 客户: 主动发起通信的应用程序
  - ① 每次执行都与服务器联系
  - ② 容易构建, 往往不需要系统特权
  - ③ 属于常规的网络应用程序, 如浏览器
- 服务器: 等待接收客户通信请求的程序
  - ① 接收客户的请求
  - ② 执行必要的操作
  - ③ 返回结果给客户

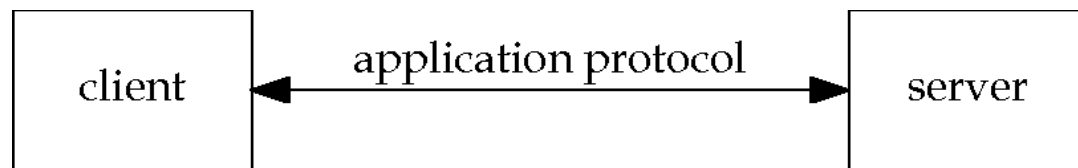


Figure 1.1 Network application: client and server.

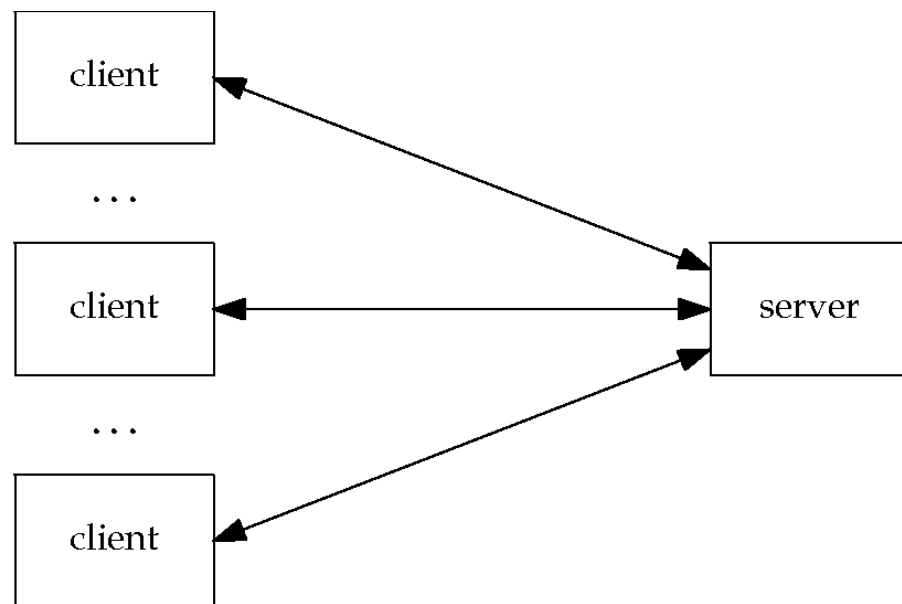


Figure 1.2 Server handling multiple clients at the same time.



## 7.1.3 进程寻址

### □ 计算机网络的基础知识

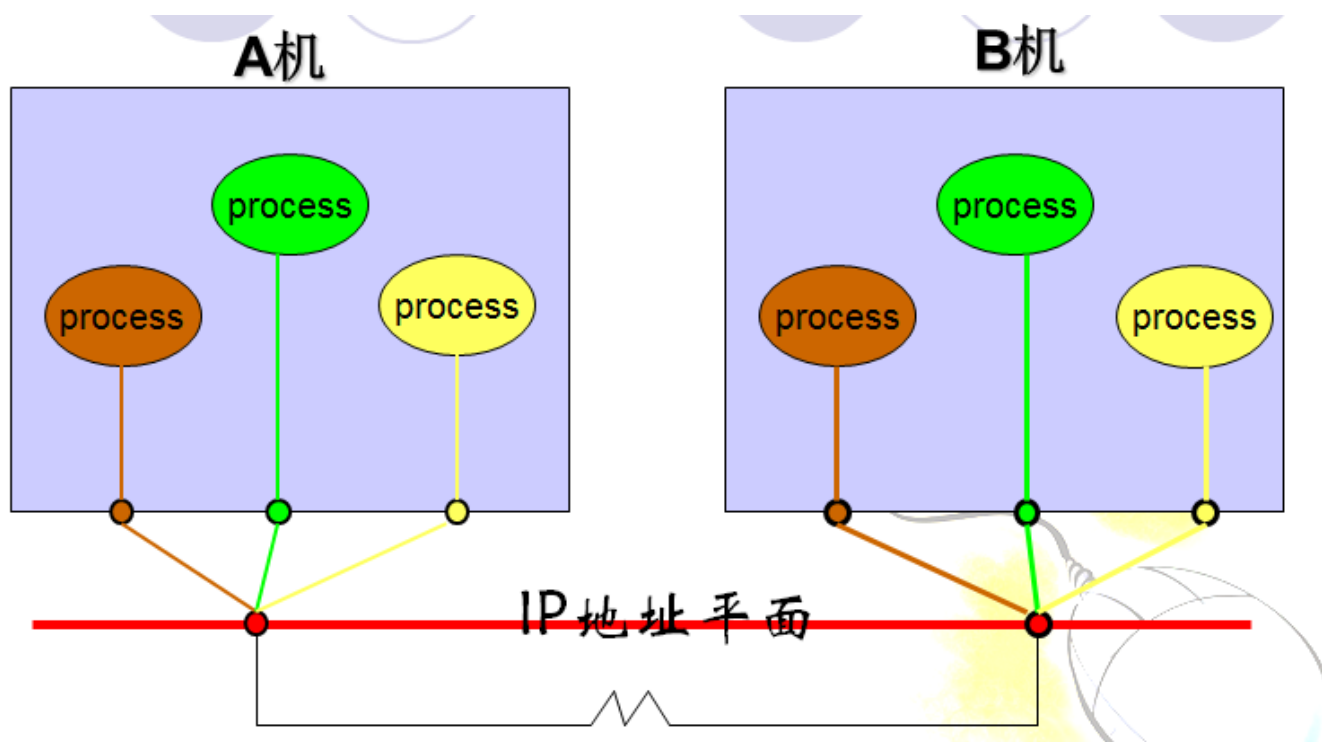
- 我们将学习采用Berkeley套接字编写C/S架构的网络应用程序
- 客户端进程-服务器进程
- 进程间的寻址问题(TCP\UDP)

采用TCP协议的进程间寻址

(源IP, 源port, 目的IP, 目的port)

采用UDP协议的进程间寻址

(目的IP, 目的port)



## 7.1.4 网络编程基础-字节序

□ 字节：计算机上传输和存储信息的最小单位

○ 其他各种数据类型，都是由字节构成

int: 4字节; short: 2字节; long: 4字节; char: 1字节;  
float: 4字节; unsigned int: 4字节; double: 8字节;  
char(\*): 指针变量, 4字节(32位CPU), 8字节(64位CPU)

□ 字节序(Byte Ordering): 指多字节数据在计算机内存中存储（或传输）时各字节的存储（或传输）顺序。不同的CPU和操作系统平台上存储数据的方式不同，包括小端（Little endian）和大端（Big endian）。

## 7.1.4 网络编程基础-主机字节序

❑ 大端存储模式(Big-Endian): 将高序字节存储在低地址单元

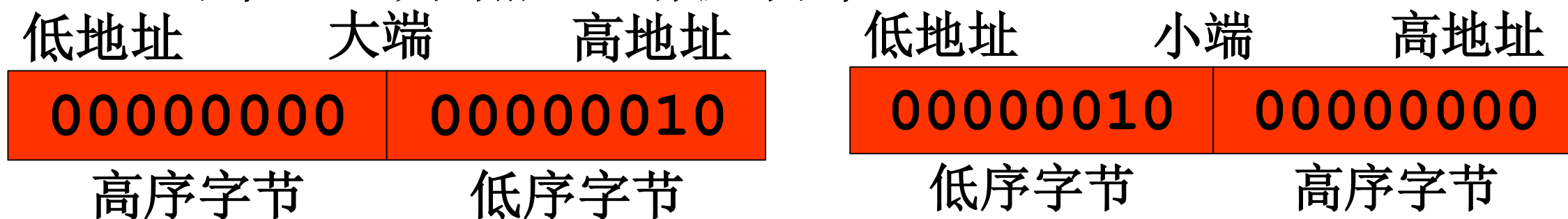
○ 以unsigned short类型的数据0x0002 (2字节) 为例:

- 高序字节0x00放在低位地址标识的字节
- 低序字节0x02放在高位地址标识的字节

❑ 小端存储模式(Little-Endian): 将低序字节存储在低地址单元

○ 以unsigned short类型的数据0x0002为例

- 字节0x00放在高位地址标识的字节
- 字节0x02放在低位地址标识的字节



## 7.1.4 网络编程基础-主机字节序

- 不同CPU和OS的主机字节序如下：

CPU	OS	字节序
Intel x86	全部	小端
MIPS	NT	小端
MIPS	UNIX	大端
ARM	全部	大/小端

请同学们尝试写一个程序检测本机的字节序是小端还是大端(Little-Endian.c)

在X86下，所有操作系统都采用小端字节序来存储和读取数据

## 课堂练习

假设在 0x20 号开始的地址中保存4字节int型数据 0x12345678，大端序 CPU 保存方式如下图所示：

0x20号	0x21号	0x22号	0x23号
0x12	0x34	0x56	0x78

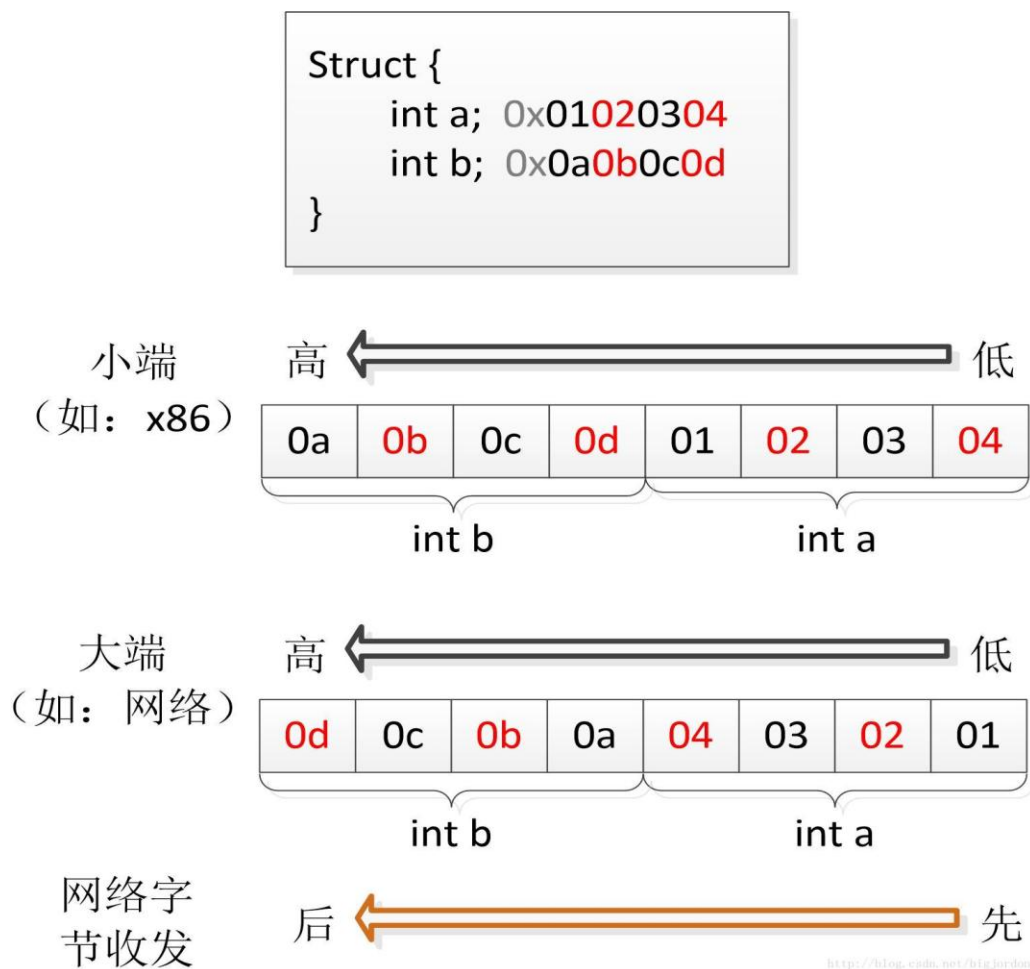
小端序 CPU 保存方式如下图所示：

0x20号	0x21号	0x22号	0x23号
0x78	0x56	0x34	0x12

## 7.1.4 网络编程基础-网络字节序

### □ 网络字节序(Network-Endian)

- 网络字节顺序是TCP/IP中规定好的一种数据表示格式，它与具体的CPU类型、操作系统等无关，从而可以保证数据在不同主机之间传输时能够被正确解释。**在发送数据前，要将数据转换为网络字节序。**
- 网络字节顺序采用大端(Big-Endian)排序方式，总是从低位地址开始传输。
- 发送数据包时，程序将主机字节序转换为网络字节序；接收数据包时，则将网络字节序转换为主机字节序。



## 7.1.4 网络编程基础-主机字节序

### □ 套接字API提供字节序的相互转化

- ① htons: 主机字节序转化为网络字节序
- ② ntohs: 网络字节序转化为主机字节序
- ③ htonl: 主机字节序转化为网络字节序
- ④ ntohl: 网络字节序转化为主机字节序

○ < netinet/in.h >

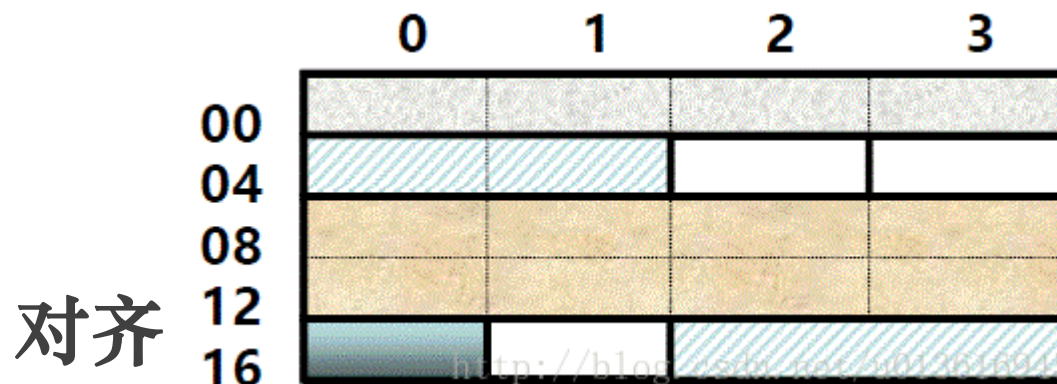
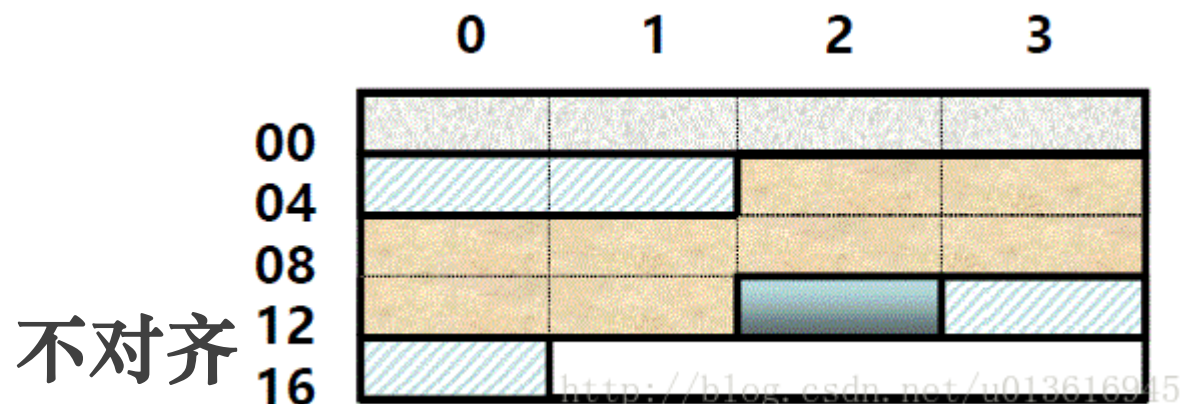
○ <endian.h> Linux下



## 7.1.5 网络编程基础—数据对齐

- 现代计算机中内存空间都是按照字节(byte)划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定变量的时候经常在特定的内存地址访问，这就需要各类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排列，这就是对齐。

假设在一个32位的系统中，有 `int i; short k; double x; char c; short j;` 几个变量，如果我们不进行数据对齐的话，存放这5个变量需要**17字节**。如果进行数据对齐的话，虽然需要**20字节**存放变量。



## 7.1.5 网络编程基础—数据对齐

□ 默认对齐规则

□ 数据类型对齐值：

char型数据自身对齐值为1

short为2，int、float为4，double为8(windows)

解释：char变量只要有一个空余的字节即可存放；short要求首地址能被2整除；int、float要被4整除，double要被8整除

□ **结构体的对齐值：其成员中自身对齐值最大的那个值。**

解释：结构体最终对齐按照数据成员中最长的类型的整数倍

每个成员都有自己的对齐值，这个值小于最大对齐值

GCC下：struct my{ char ch; int a;} （1字节+4字节）->（4字节+4字节）  
sizeof(int)=4; sizeof(my)=8; （非紧凑模式）

## 7.1.5 网络编程基础—数据对齐

### 结构体对齐的示例

```
struct A
{
    char x1; //1 byte
    char x2[5]; //5 bytes
    int x3; //4 bytes
};
```

x1	x2[0]	x2[1]	x2[2]
x2[3]	x2[4]		
x3	x3	x3	x3

sizeof(struct A)  
= 12

```
struct my
{
    char ch; //1 byte
    int a; //4 bytes
};
```

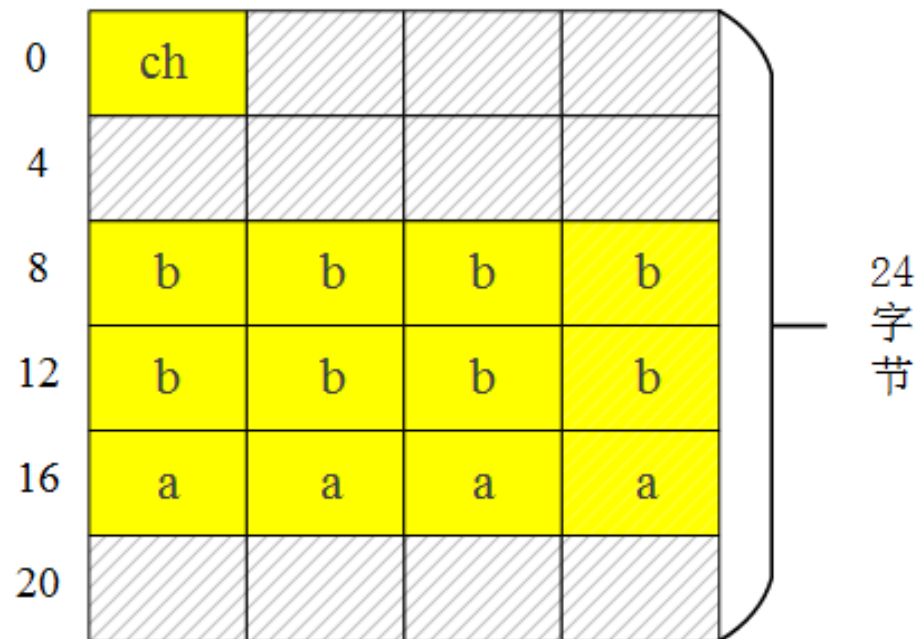
ch			
a	a	a	a

sizeof(struct my)  
= 8

## 7.1.5 网络编程基础—数据对齐

### 课堂练习

```
struct my
{
    char ch; //1 byte
    double b; // 8 bytes
    int a; //4 bytes
};
```



结构体最终对齐按照数据成员中最长的类型的整数倍，因此必须是8字节的整数倍

## 7.1.5 网络编程基础—数据对齐

在编写应用层程序时，通常采用结构体来定义各种头部(IP\UDP\TCP等)，地址表示方式等，而C语言中会对结构体进行对齐，因此可能实际占用的地址空间与结构体中变量类型对应的字节总和有差异(填充)。因此在编写网络程序时，**需要仔细设计结构体，以及使用sizeof()来获得结构体的大小。**

```
connect(s, (struct sockaddr *)&sin, sizeof(sin));
```

```
bind(msock, (struct sockaddr *)&sin, sizeof(sin));
```

## 7.1.6 网络编程基础—DNS

- ❑ 层次名字空间
- ❑ 便于记忆和使用
- ❑ 计算机通信时无法使用
- ❑ 域名地址解析：
  - 名字到IP地址的解析 (gethostbyname)
  - IP地址到域名的解析 (gethostbyaddr)

在编写客户端程序时，如果是通过**域名指定服务器地址**，则需要调用gethostbyname来设置；如果是通过点分十进制表示的IP地址(字符串)来指定服务器地址，则需要调用gethostbyaddr来设置。

## 7.1.7 网络编程基础——标准/非标准的应用协议

- 标准的应用层协议
  - 属于TCP/IP协议簇
  - 文件传送, 电子邮件, 远程登录等
- 非标准的应用层协议
  - 使用TCP/IP进行通信的程序
  - 标准化之后, 会成为标准协议
- TCP/IP定义了许多标准的应用服务
  - TELNET, SMTP, POP, HTTP, FTP



## 7.1.8 网络编程基础—客户端的参数化

□ 提高通用性的客户软件：可以指定服务器地址，也可以指定服务器的服务端口（全参数化客户：fully parameterized client）

□ 参数化

- 远程主机地址(服务器)
- 端口号
- 其它可选参数
- 便于测试

```
int main(int argc, char* argv[])
```

通过主函数的参数来传递这些信息

```
<程序名> <IP> <PORT>
```

```
srv_addr.sin_addr.s_addr = inet_addr(argv[1]);  
srv_addr.sin_port = htons(atoi(argv[2]));
```

## 7.1.9 网络编程基础—无连接/面向连接服务器

### □ UDP: 无连接交互

- 没有可靠保证
- 依赖下层系统保证
- 程序中应该有相应保障措施

在socket套接字函数组中，  
为TCP和UDP分别设计了不同的API函数

### □ TCP: 面向连接的交互

- 提供传输可靠性
- 程序要求简单

### □ 选用UDP的情况

- 下层系统可靠性（例如在局域网环境）
- 应用要求
- 广播或者组播

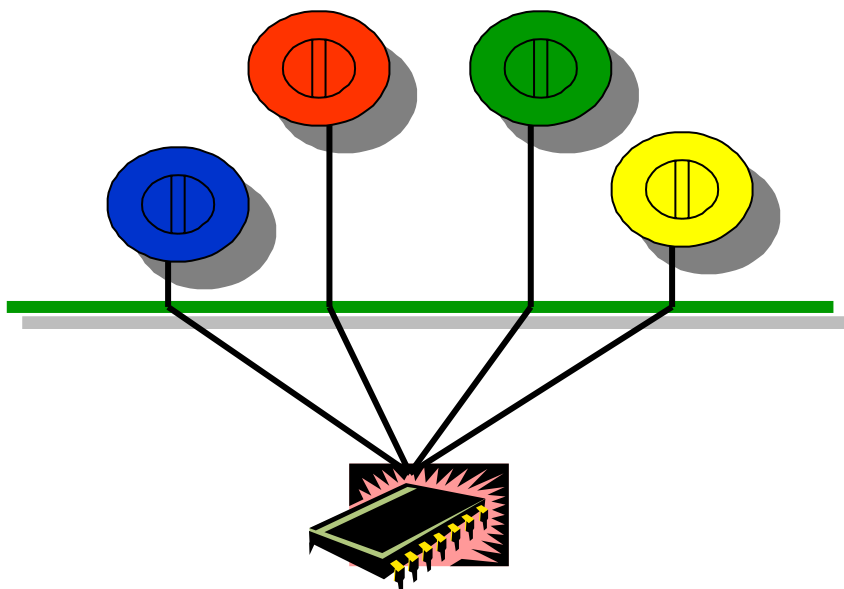
## 7.1.10 客户/服务器软件中的并发处理

- ❑ 并发的概念
- ❑ 网络中的并发
- ❑ 服务器软件中的并发
- ❑ 客户软件中的并发
- ❑ 操作系统的并发功能
- ❑ 一个并发例子程序

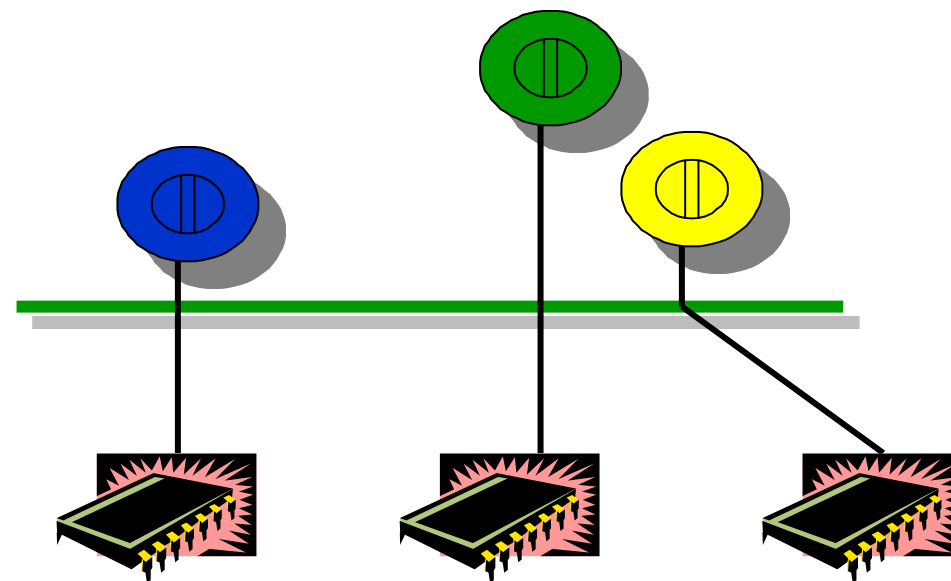


## 7.1.10 网络编程基础—并发的概念

- 并发有真正的并发(并行: Parallelism)和表面上的并发(并发: Concurrency) (一般采用分时机制)



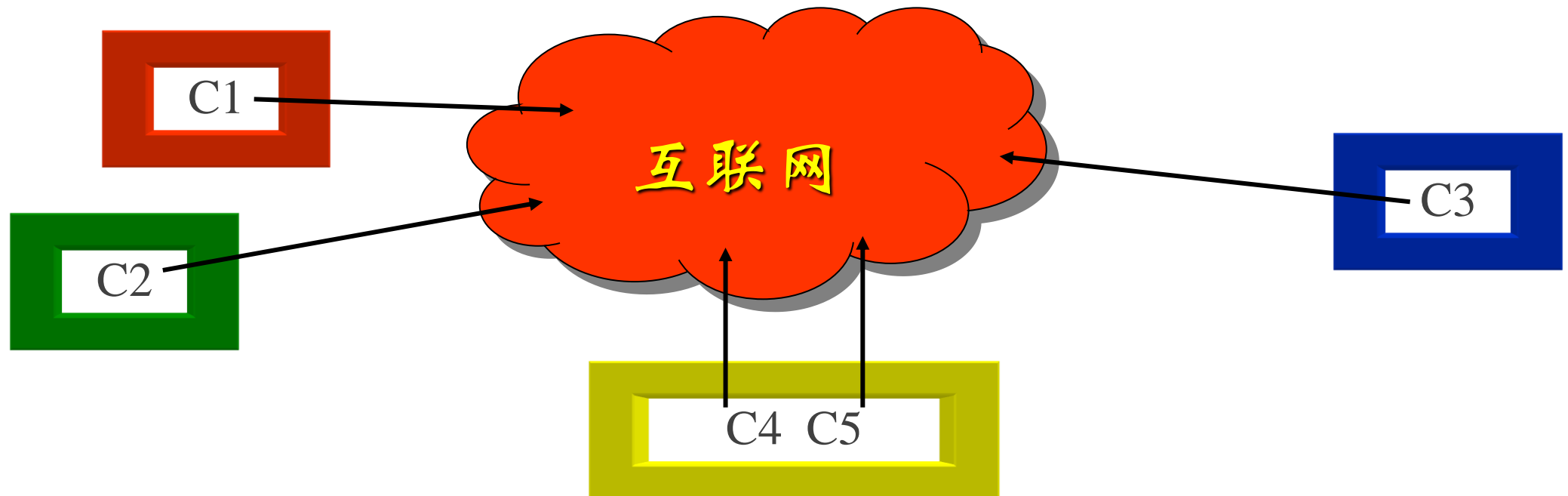
并发模型



并行模型

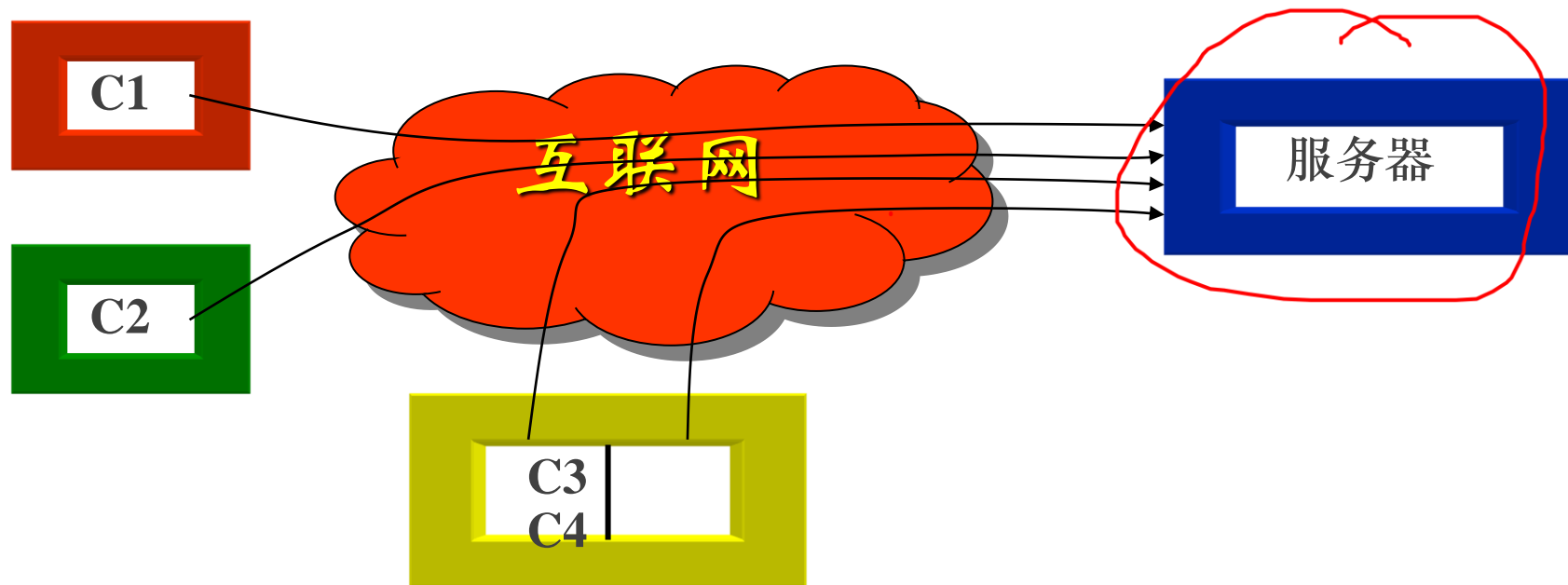
## 7.1.10 网络编程基础—网络中的并发

- 单个网络各个机器之间许多成对进程好像独立使用网络资源（通道，机器等）
- 一个计算机系统中存在并发（分时）
- 一组机器上所有的客户之间存在并发



## 7.1.10 网络编程基础—服务器中的并发

- ❑ 单个服务器必须并发处理多个传入请求
- ❑ 并发服务器可以让多个远程用户同时使用服务，实现起来比较复杂
- ❑ 其余部分主要介绍术语和概念；涉及其它问题：算法，设计，运行规则等在后续各章介绍



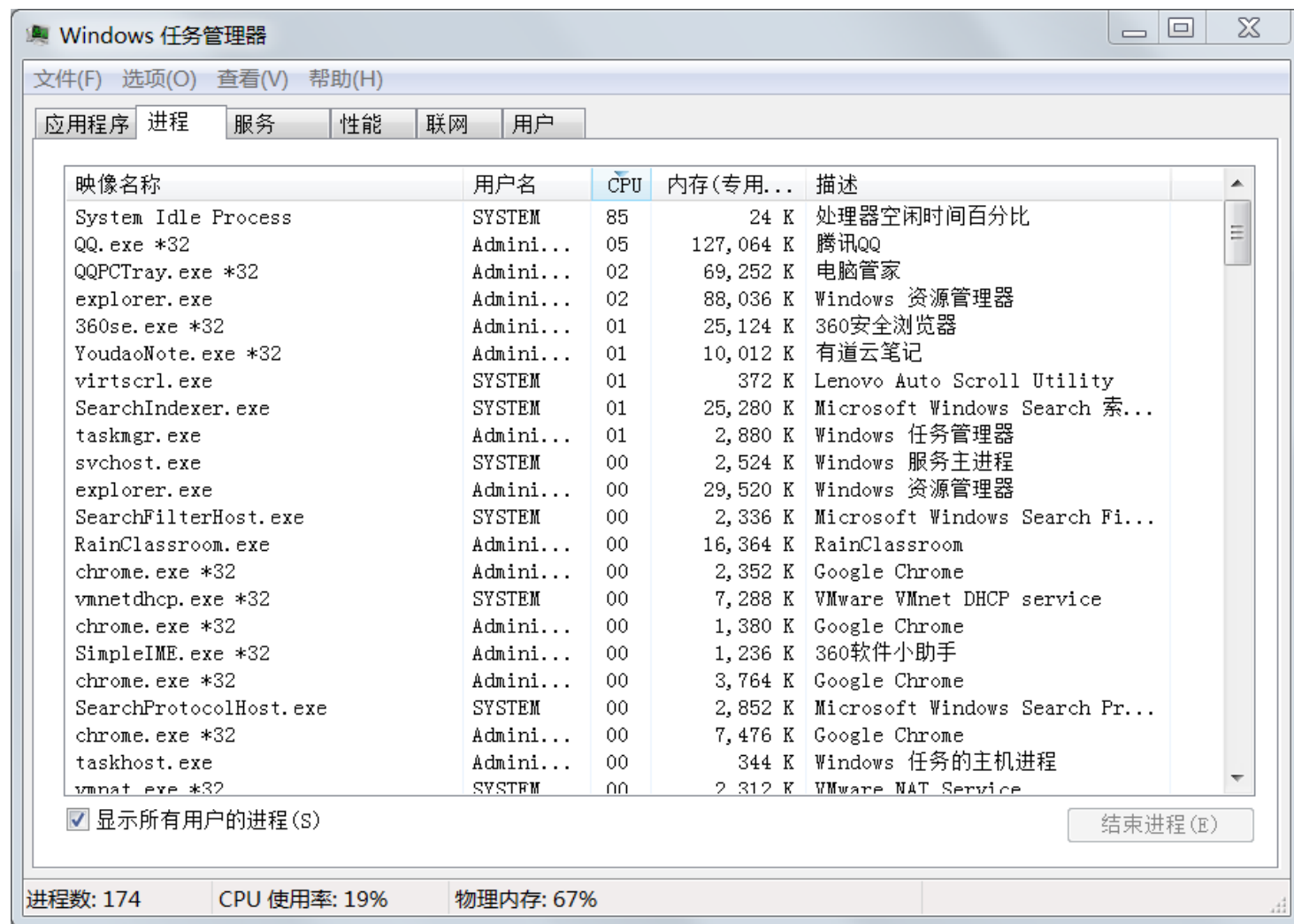
## 7.1.10 网络编程基础—客户软件的并发

- 要使客户软件并发执行，一般并不需要程序员为此特别花功夫。
- 现代操作系统一般允许用户并发地执行客户程序



## 7.1.10 网络编程基础——操作系统的并发功能

- 多进程操作系统
- 进程的概念：进程定义了一个计算的基本单元，它是一个执行某一个特定程序的实体，它拥有独立的地址空间、执行堆栈、文件描述符等。



## 7.1.10 网络编程基础—创建进程

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

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

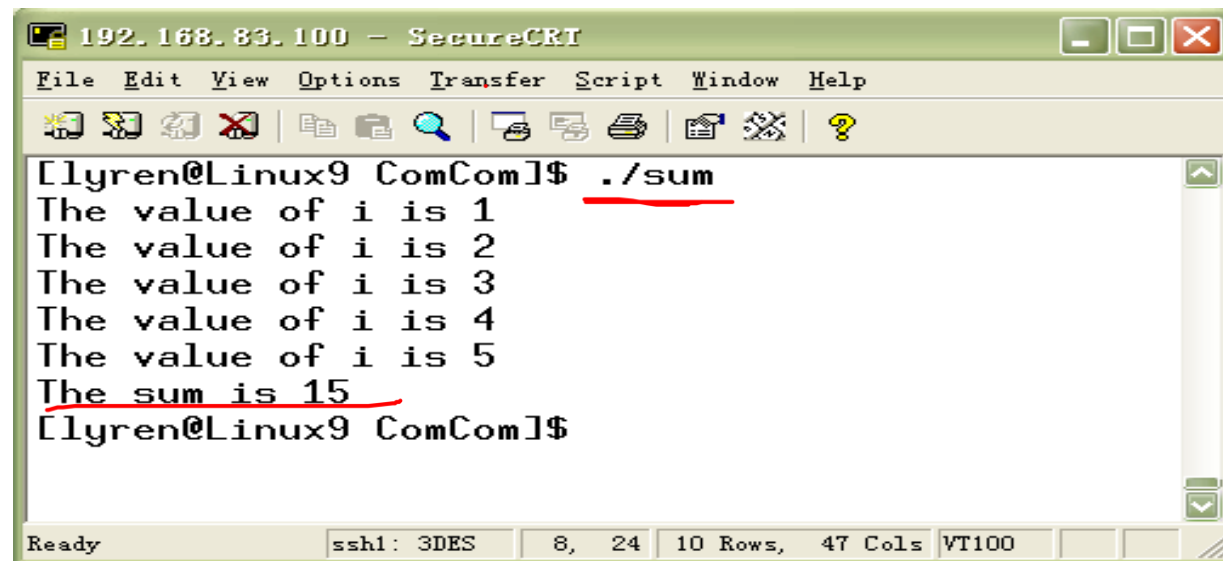
```
pid_t fork(void)
```

- ❑ 返回：父进程中返回子进程的进程ID，子进程返回0，-1—出错
- ❑ fork后，父子进程共享数据空间、代码空间、堆栈、所有的文件描述字。

所有多进程编程的代码参考：第1讲-网络编程基础\多进程

## 7.1.10 网络编程基础——一个顺序执行的C实例

```
#include <stdlib.h>
#include <stdio.h>
int sum;
void main(void)
{
    int i;
    sum = 0;
    for(i=1; i<=5; i++)
    {
        printf("The value of i is %d\n",i);
        fflush(stdout);
        sum += i;
    }
    printf("The sum is %d\n",sum);
}
```



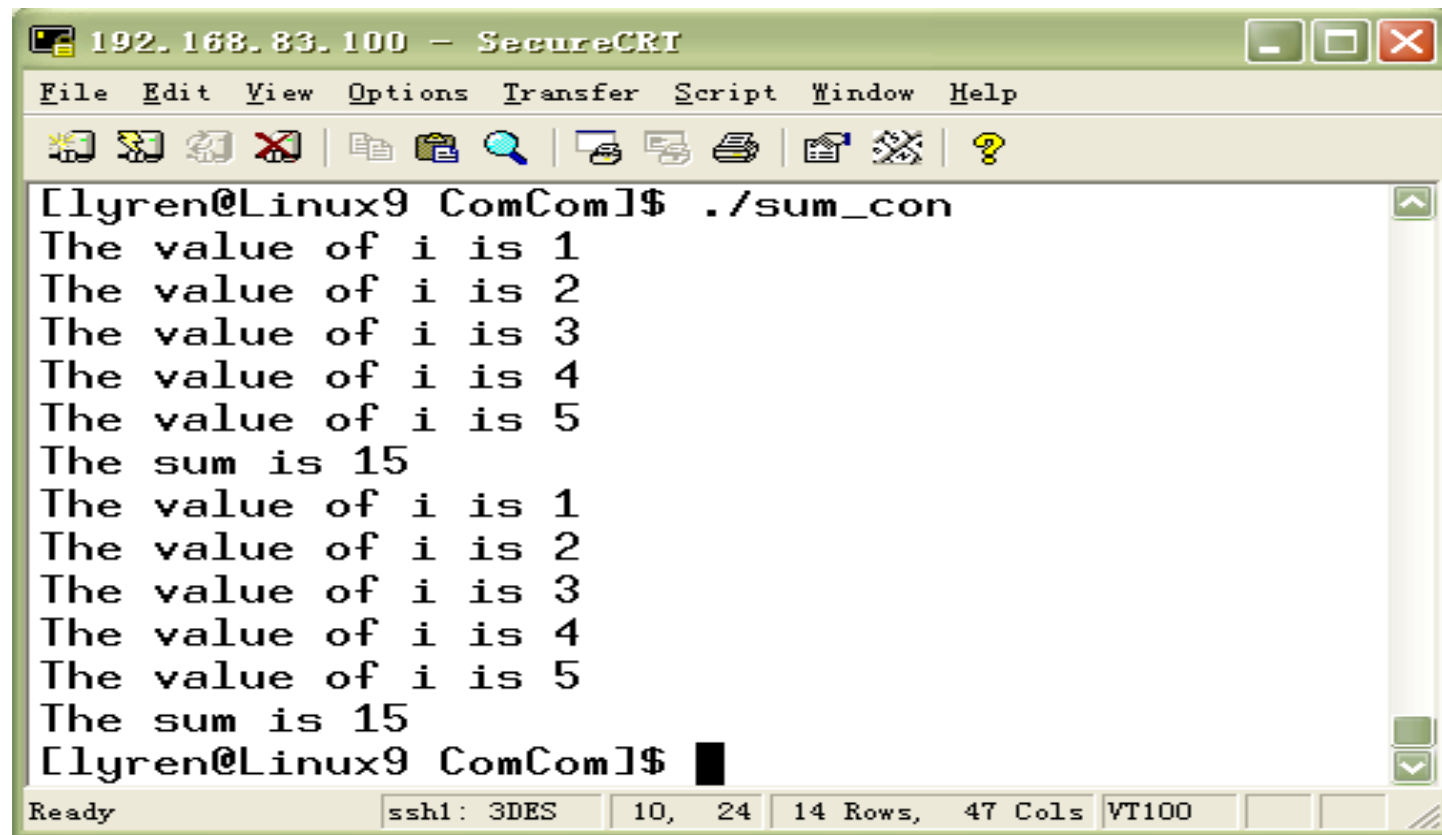
The screenshot shows a SecureCRT terminal window with the title bar "192.168.83.100 - SecureCRT". The menu bar includes File, Edit, View, Options, Transfer, Script, Window, and Help. The command prompt shows the user [llyren@Linux9 ComCom] running the command ./sum. The output of the program is displayed as follows:

```
[llyren@Linux9 ComCom]$ ./sum
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The sum is 15
[llyren@Linux9 ComCom]$
```

The status bar at the bottom indicates "Ready", "ssh1: 3DES", "8, 24", "10 Rows, 47 Cols", and "VT100".

## 7.1.10 网络编程基础—fork()

```
#include <stdlib.h>
#include <stdio.h>
int sum;
int main(void)
{
    int i;
    sum = 0;
    fork();
    for(i=1; i<=5; i++)
    {
        printf("The value of i is %d\n",i);
        fflush(stdout);
        sum += i;
    }
    printf("The sum is %d\n",sum);
    exit(0);
}
```



```
192.168.83.100 - SecureCRT
File Edit View Options Transfer Script Window Help
[lyren@Linux9 ComCom]$ ./sum_con
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The sum is 15
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The sum is 15
[lyren@Linux9 ComCom]$
```

Ready ssh1: 3DES 10, 24 14 Rows, 47 Cols VT100

```

#include <stdlib.h>
#include <stdio.h>
int mul,sum;
int main(void) {
    int i,pid;
    sum = 0;
    mul = 1;
    if((pid=fork()) > 0) {
        for(i=1; i<=5; i++) {
            printf("The value of i is %d\n",i);
            fflush(stdout);
            sum += i; sum=sum+1;
        }
        printf("The sum is %d\n",sum);
    }
    else if (pid == 0) {
        for (i=1; i<=5; i++) {
            printf("The value of i is %d\n",i);
            fflush(stdout);
            mul *= i; mul=mul*I;
        }
        printf("The multiplex is %d\n",mul);
    }
    exit(0);
}

```

```

192.168.83.100 - SecureCRT
File Edit View Options Transfer Script Window Help
[lyren@Linux9 ComCom]$ ./mul_con
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The multiplex is 120
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The sum is 15
[lyren@Linux9 ComCom]$
Ready ssh1: 3DES 14, 24 16 Rows, 47 Cols VT100

```

另一并发版本

## 7.1 网络编程基础小结

- ❑ TCP/IP协议栈：网络应用程序是针对应用层，底层都在OS中
- ❑ C/S架构：需要编写客户端程序和服务器端程序
- ❑ 主机字节序和网络字节序：(重点)
- ❑ 结构体对齐：sizeof()
- ❑ DNS：通过socket API来设置服务器IP地址
- ❑ 标准和非标准应用程序：
- ❑ 客户软件参数化：通过main函数的参数传递
- ❑ UDP/TCP：SOCK\_STREAM\SOCK\_DGRAM
- ❑ 网络并发：fork()

## 7.2 套接字基础

### □ 介绍套接字函数的设计

- TCP/IP套接字运行于OS中，只为网络应用程序提供调用接口(API)
- 为了使API可以适合于各种硬件和操作系统，参数中不应该与硬件和OS相关，因此需要进行主机字节序和网络字节序转换、需要sizeof()来设置结构体的size
- 套接字API是一组功能定义，在每个不同的OS中，可以具体定义

### □ Berkeley套接字

- 套接字的设计来自I/O函数的扩展，通过套接字描述符来发送/接收数据
- 常用套接字函数



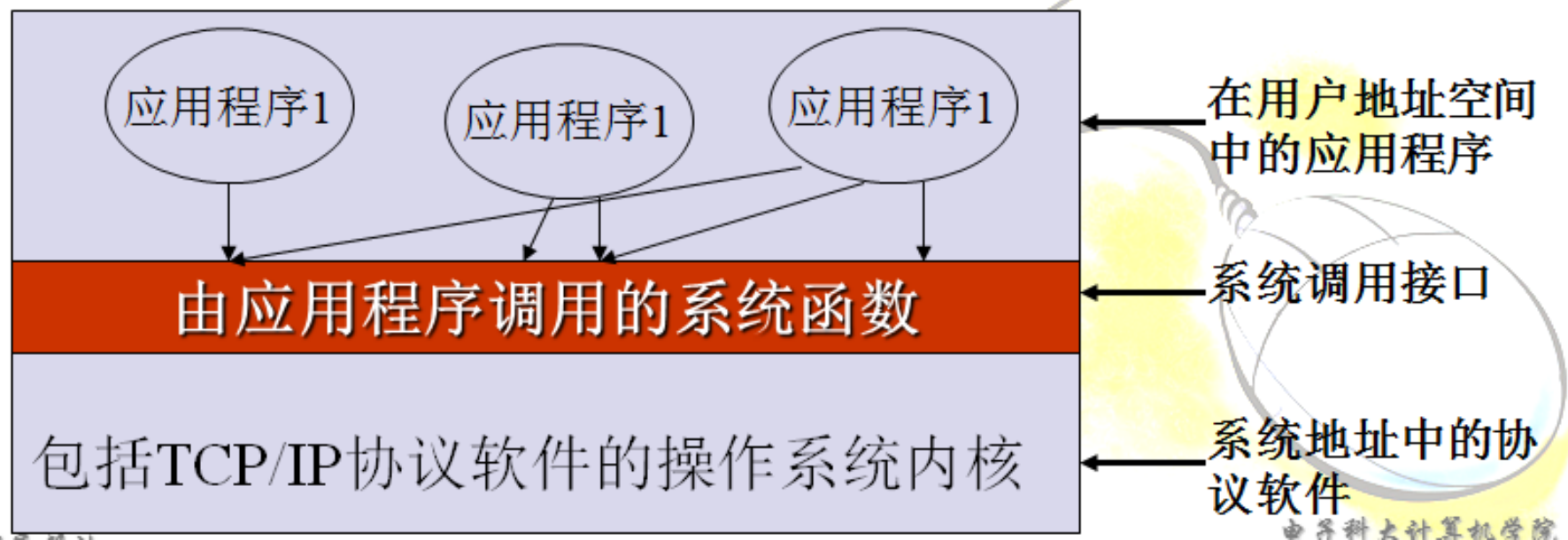
## 7.2.1 套接字基础—API接口功能

- ❑ 分配用于通信的本地资源
- ❑ 指定本地和远程通信端点  
(源IP、目的IP)
- ❑ (客户端) 启动连接
- ❑ (客户端) 发送数据报
- ❑ (服务器端) 等待连接到来  
(listen()/accept())
- ❑ 发送或者接收数据
- ❑ 判断数据何时到达
- ❑ 产生紧急数据
- ❑ 处理到来的紧急数据
- ❑ 从容终止连接
- ❑ 处理来自远程端点的连接终止
- ❑ 异常终止通信
- ❑ 处理错误条件或者连接异常终止
- ❑ 连接结束后释放本地资源

关于套接字函数设计的思考

## 7.2.2 套接字基础—系统调用

- 系统调用：应用程序和操作系统传递控制权（函数调用）
- 目的是从操作系统获得服务
- 一定的权限控制



关于套接字函数设计的思考

## 7.2.3 套接字基础—网络通信的两种基本方法

### □ 使用新的系统调用来访问TCP/IP

- 对于每个概念性的操作实现一个系统调用
- 创建新的系统调用并不明智

### □ 使用一般的I/O调用来访问TCP/IP

- 使用一般的I/O调用，但是进行了扩充，既可以用于I/O，又可以用于网络协议

### □ 混合方法

- 尽可能使用基本的I/O功能
- 增加一些函数来实现其它操作

关于套接字函数设计的思考

## 7.2.4 套接字基础—Linux中提供的基本I/O功能

□ 理解基本I/O如何扩展功能，六个基本的系统函数：

操作	含义
Open	为输入或输出操作准备一个设备或者文件
Close	终止使用以前已打开的设备或者文件
Read	从输入设备或者文件中得到数据
Write	数据从应用程序存储器传到设备或文件中
Lseek	转到文件或者设备中的某个指定位置
Ioctl	控制设备或者用于访问该设备的软件

关于套接字函数设计的思考

## 7.2.4 套接字基础—基本I/O举例

int desc; // 访问一个文件需要定义一个文件描述符  
desc=open(“filename”, O\_RDWR, 0) // 初始化文件描述符  
read(desc,buffer,128) // 通过文件描述符读/写数据  
close(desc) // 使用完后需关闭文件描述符

### 关于套接字函数设计的思考

```
#include <fcntl.h>
```

```
int open(const char *pathname, int oflag, ... /* mode_t mode */);
```

返回值：成功则返回 文件描述符，否则返回 -1

代码参考-第2讲-套接字基础  
No.2/copy.c

## 7.2.4 套接字基础—将Linux I/O用于TCP/IP

- 扩展文件描述符：可以用于网络通信
  - 扩展read和write：可以用于网络标识符
- read类比于从网络接收数据 / write类比于向网络发送数据
- 额外功能的处理，增加新系统调用：
    - 指明本地和远端的端口，远程IP地址
    - 使用TCP还是UDP
    - 启动传输还是等待传入连接
    - 可以接受多少传入连接
    - 使用UDP传输数据

## 7.2.5 套接字基础—Berkeley套接字

### □ 什么是套接字

- TCP/IP协议存在于OS中，网络服务通过OS提供
- 在OS中增加支持TCP/IP的**系统调用**——Berkeley套接字
- 如Socket, Connect, Send, Recv等函数
- 1981提出BSD4.1 UNIX, 基于BSD4.4版本

### □ Berkeley套接字

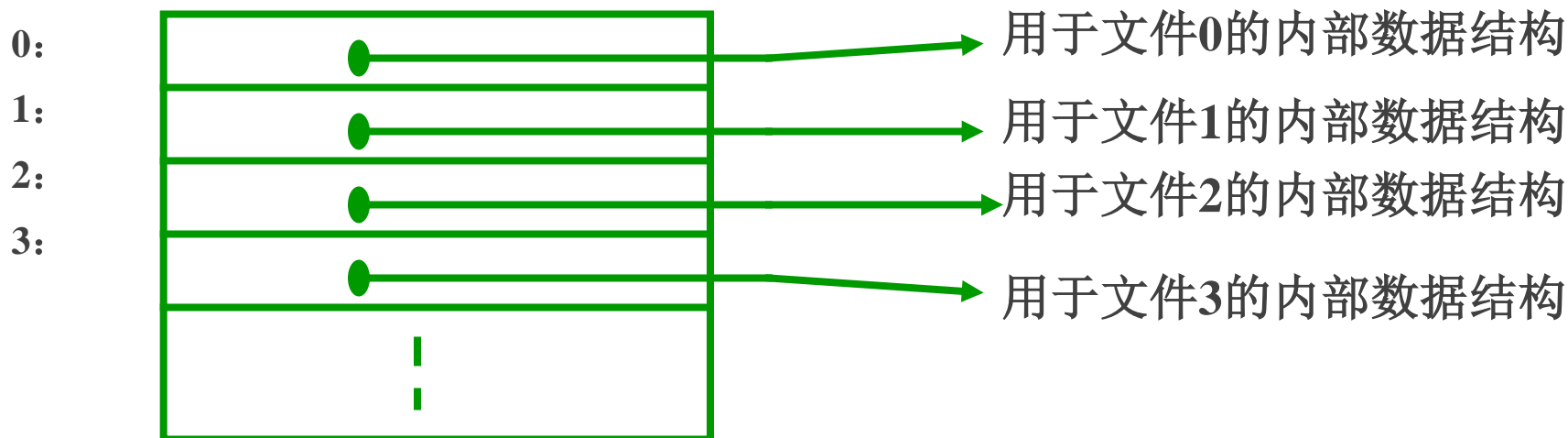
- ARPA要求伯克利分校将TCP/IP移植到UNIX中
- 需要创建一个接口，便于应用程序使用这个接口进行网络通信
- 尽可能使用现有的系统调用，同时添加新的系统调用支持TCP/IP
- 这个系统被称为BSD UNIX套接字，成为事实上的标准

## 7.2.6 套接字基础—套接字描述符

- ❑ OS将文件描述符实现为一个指针数组，指向一个内部的数据结构：进程描述符表的下标
- ❑ 套接字和文件类似，每个活动套接字使用一个小整数标识，进程的文件描述符和套接字描述符值不能相同
- ❑ socket函数：创建套接字描述符（不是open函数）

进程的文件描述符表

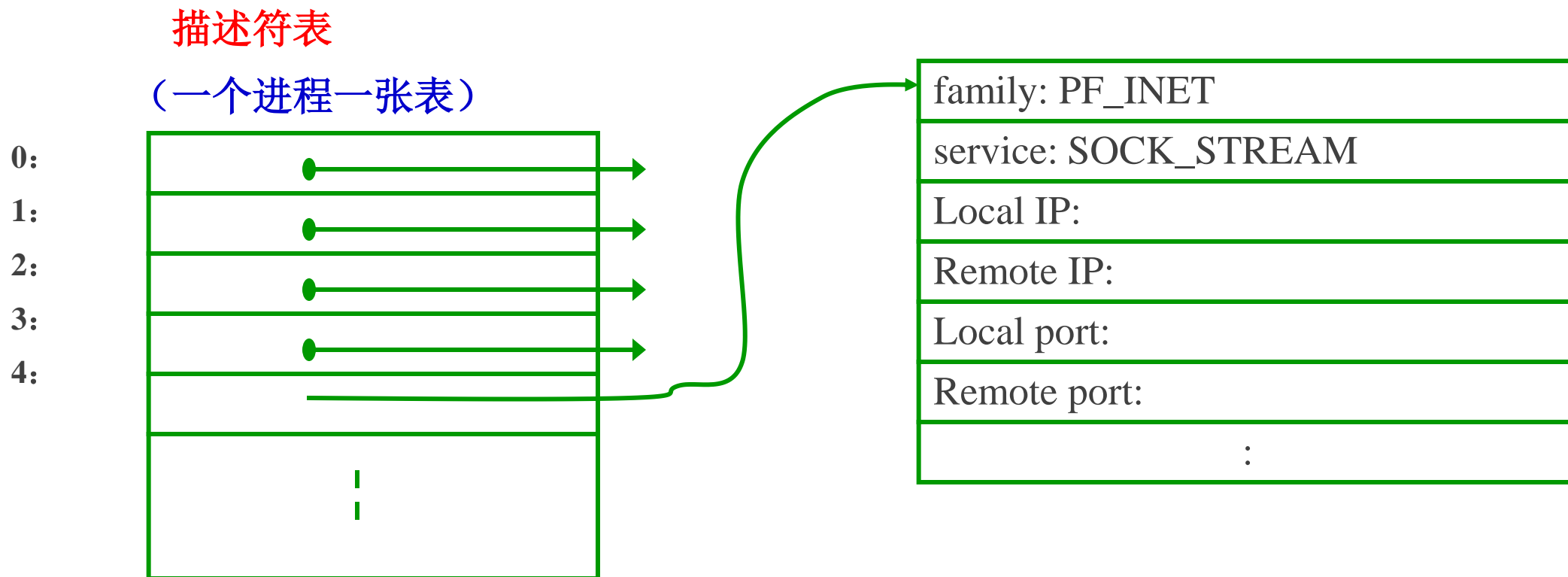
（一个进程一张文件描述符表）





## 7.2.6 套接字基础——针对套接字的系统数据结构

- 调用socket将创建一个新的描述符条目
- 结构的许多字段是其他的系统调用来填



## 7.2.7 套接字基础——主动套接字和被动套接字

- 创建方式相同，使用方式不同
- 等待传入连接的套接字——**被动**，如服务器套接字
- 发起连接的套接字——**主动**，如客户套接字
- 指明端点地址：创建时不指定，使用时指明
  - TCP/IP需要指明协议端口号和IP地址
  - TCP/IP协议族：**PF\_INET** (IPv4协议)
  - TCP/IP的地址族：**AF\_INET** (IPv4地址)

## 7.2.8 套接字基础—类属地址结构

- 套接字系统定义的一般化的地址结构
  - (地址族, 该族的端点地址)

### 套接字的普通C定义结构

```
struct sockaddr {  
    [u_char  sa_len; //长度]  
    u_short  sa_family; //2B地址类型  
    char  sa_data[14]; //14B地址值  
} 通用的地址结构 (只是很适用于  
AF_INET族中的地址)
```

sizeof(sockaddr)=16字节  
sizeof(sockaddr\_in)=16字节

### TCP/IP的地址定义

```
struct sockaddr_in  
{  
    [u_char  sin_len; //长度]  
    u_short  sin_family; //2B地址类型  
    u_short  sin_port; //2B协议端口号  
    struct in_addr  sin_addr; //4B, IP地址  
    char  sin_zero[8]; //8B未使用设置为0  
} IP专用的结构
```

早期sockaddr只包含2个字节的sa\_family和后面的14个字节的data, 后来为了OSI兼容性, 第一个字节变为长度len, 值为16, 相应的, family就变为1个字节;

## 7.2.9 套接字基础—套接字API的主要系统调用

### □ 套接字调用分为两组：

- 主调用：提供对下层功能的访问
- 实用例程：提供帮助

### □ 套接字带有参数，允许以多种方式来使用它们。

- 可被客户或服务器使用
- 可被TCP或UDP使用
- 可被特定或非特定的远程端点地址所使用

## 7.2.9 套接字基础—套接字API中的主要系统调用

`int socket(int domain, int type, int protocol)`

❑ 功能：创建一个新的套接字，返回套接字描述符

❑ 参数说明：

○ domain：协议类型，指明使用的协议栈，如TCP/IP使用的是AF\_INET

○ type：指明需要的服务类型，如

• SOCK\_DGRAM：数据报服务，UDP协议

• SOCK\_STREAM：可靠数据流服务，TCP协议

• SOCK\_RAW：低于传输层的低级协议或物理网络提供的套接字类型，可以访问内部网络接口。（不要求）

○ protocol：一般都取0

❑ 返回值

❑ 当套接字创建成功时，返回套接字，失败返回“-1”，错误代码则写入errno中

举例：s=socket(AF\_INET, SOCK\_STREAM, 0)

## 7.2.9 套接字基础—套接字API中的主要系统调用

`int connect(int sockfd, struct sockaddr *server_addr, int sockaddrlen)`

- ❑ 功能：同远程服务器建立主动连接，成功时返回0，若连接失败返回-1。
- ❑ 参数说明：
  - sockfd：套接字描述符，指明创建连接的套接字
  - server\_addr：指明远程端点：IP地址和端口号
  - sockaddrlen：地址长度
- ❑ 返回值：
  - 成功返回0，失败返回-1。

```
connect(s, (struct sockaddr *)&sin, sizeof(sin));
```

## 7.2.9 套接字基础—套接字API中的主要系统调用

`int send(int sockfd, const void * data, int data_len, unsigned int flags)`

### □ 功能:

- 在TCP连接上发送数据，返回成功传送数据的长度，出错时返回-1。
- send会将外发数据复制到OS内核中，也可以使用send发送面向连接的UDP报文。

### □ 参数说明:

- sockfd: 套接字描述符
- data: 指向要发送数据的指针
- data\_len: 数据长度
- flags: 一直为0

### □ 返回值

- 返回值: 失败时，返回值小于0；超时或对端主动关闭，返回值等于0；成功时，返回值是返回发送数据的长度。

举例(p50): `send(s, req, strlen(req), 0);`

## 7.2.9 套接字基础—套接字API中的主要系统调用

`int sendto(int sockfd, const void * data, int data_len, unsigned int flags,  
struct sockaddr *remaddr, int remaddr_len)`

- ❑ 功能：基于UDP发送数据报，返回实际发送的数据长度，出错时返回-1
- ❑ 参数说明：
  - sockfd: 套接字描述符
  - data: 指向要发送数据的指针
  - data\_len: 数据长度
  - flags: 一直为0
  - remaddr: 远端地址：IP地址和端口号
  - remaddr\_len: 地址长度

例： `sendto(sockfd, buf, sizeof(buf), 0, (struct sockaddr *)&address, sizeof(address));`



## 7.2.9 套接字基础—套接字API中的主要系统调用

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

### □ 功能:

- 从TCP接收数据，返回实际接收的数据长度，出错时返回-1。
- 服务器使用其接收客户请求，客户使用它接受服务器的应答。如果没有数据，将阻塞，如果收到的数据大于缓存的大小，多余的数据将丢弃。

### □ 参数说明:

- sockfd: 套接字描述符
- buf: 指向内存块的指针
- buf\_len: 内存块大小，以字节为单位
- flags: 一般为0

例: `recv(sockfd, buf, 8192, 0);`

## 7.2.9 套接字基础—套接字API中的主要系统调用

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

- ❑ 功能：从UDP接收数据，返回实际接收的字节数，失败时返回-1
- ❑ 参数说明：
  - sockfd: 套接字描述符
  - buf: 指向内存块的指针
  - buf\_len: 内存块大小，以字节为单位
  - flags: 一般为0
  - from: 远端的地址，IP地址和端口号
  - fromlen: 远端地址长度

例：`recvfrom(sockfd, buf, 8192, 0, (struct sockaddr *)&address, sizeof(address));`

## 7.2.9 套接字基础—套接字API中的主要系统调用

`int close(int sockfd);`

### □ 功能：

- 撤销套接字。
- 如果只有一个进程使用，立即终止连接并撤销该套接字，如果多个进程共享该套接字，将引用数减一，如果引用数降到零，则撤销它。

### □ 参数说明：

- `sockfd`: 套接字描述符

例: `close(socket_descriptor)`

## 7.2.9 套接字基础—套接字API中的主要系统调用

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

❑ 功能：为套接字指明一个本地端点地址

- TCP/IP协议使用sockaddr\_in结构，包含IP地址和端口号
- 服务器使用它来指明熟知的端口号，然后等待连接

❑ 参数说明：

- sockfd: 套接字描述符，指明创建连接的套接字
- my\_addr: 本地地址，IP地址和端口号
- addrlen: 地址长度

❑ 例：`bind(sockfd, (struct sockaddr *)&address, sizeof(address));`

## 7.2.9 套接字基础—套接字API中的主要系统调用

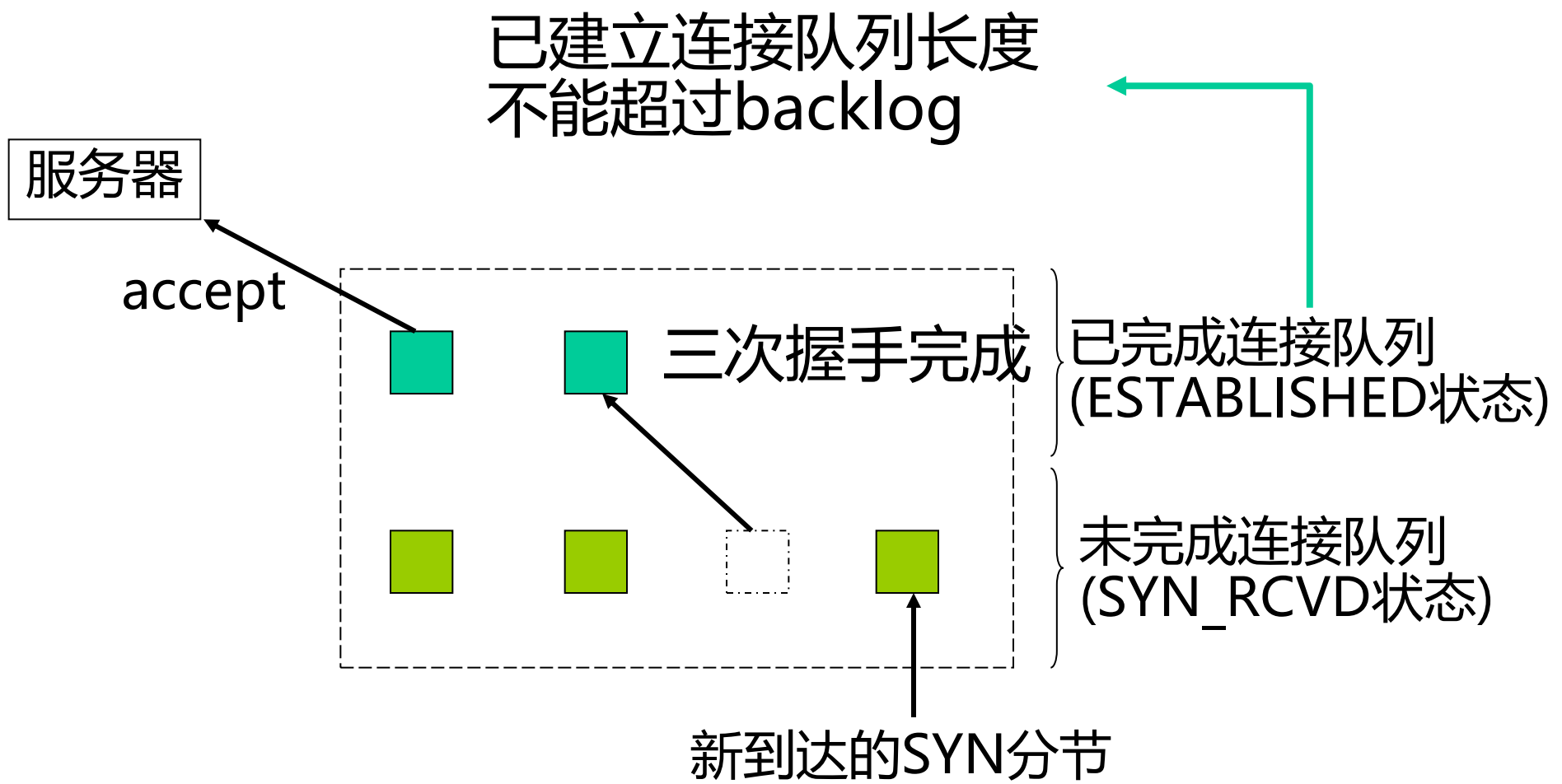
```
int listen(int sockfd, int input_queue_size)
```

### □ 功能：

- 面向连接的服务器使用它将一个套接字置为被动模式，并准备接受传入连接。用于服务器，指明某个套接字连接是被动的

### □ 参数说明：

- sockfd: 套接字描述符，指明创建连接的套接字
- input\_queue\_size: 该套接字使用的队列长度，指定在请求队列中允许的最大建立TCP连接数
- 举例：listen(sockfd, 20);



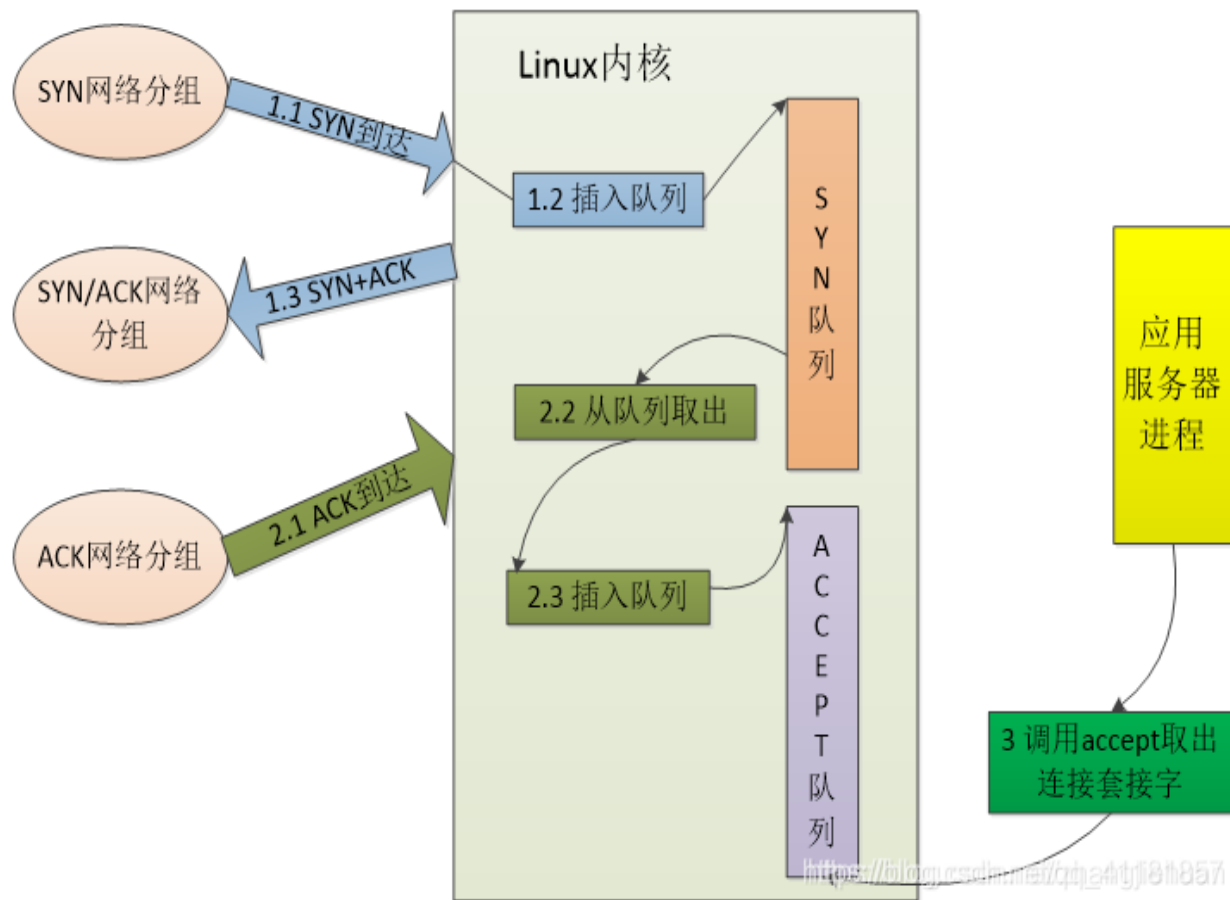
**TCP为监听套接口维护的两个队列**

# Linux的backlog

## ❑ man 2 listen

- The behavior of the backlog argument on TCP sockets changed with Linux 2.2. **Now it specifies the queue length for completely established sockets waiting to be accepted, instead of the number of incomplete connection requests.** The maximum length of the queue for incomplete sockets can be set using `/proc/sys/net/ipv4/tcp_max_syn_backlog`. When syncookies are enabled there is no logical maximum length and this setting is ignored. See `tcp(7)` for more information.
  - **If the backlog argument is greater than the value in `/proc/sys/net/core/somaxconn`, then it is silently truncated to that value; the default value in this file is 128.** In kernels before 2.4.25, this limit was a hard coded value, `SOMAXCONN`, with the value 128.
- ① backlog控制连接完成队列长度，但是限制最大长度为 `/proc/sys/net/core/somaxconn` 中的定义。
  - ② `/proc/sys/net/ipv4/tcp_max_syn_backlog` 控制未完成队列长度，如果启用 syncookies，则未完成队列长度没有限制

## 实验结果表明：服务端即使不调用accept，客户端依然可以connect成功



当客户端调用connect函数时，将引发三次握手过程，如图所示，客户端首先发送SYN请求分组，此时服务端会将请求放入SYN队列，同时向客户端发送ACK确认报文，然后客户端向服务端再次发送ACK报文。服务端收到ACK确认报文后，将SYN里的连接请求移入ACCEPT队列。此时三次握手结束，即TCP连接成功建立。然后内核通知用户空间的阻塞的服务进程，服务进程调用accept仅仅是从ACCEPT队列里取出一个连接而已。也就是说客户端调用connect连接服务器，与服务器调用accept“接受”连接是两个独立的过程

SYN队列和Accept队列都有自己的长度限制



## 7.2.9 套接字基础—套接字API中的主要系统调用

`int accept(int sockfd, void *addr, int addrlen);`

❑ 功能：获取传入连接请求，返回**新的连接的套接字描述符**。

- 为每个新的连接请求创建了一个新的套接字，服务器只对新的连接使用该套接字，原来的监听套接字接受其他的连接请求。
- 新的连接上传输数据使用新的套接字，使用完毕，服务器将关闭这个套接字。

❑ 参数说明：

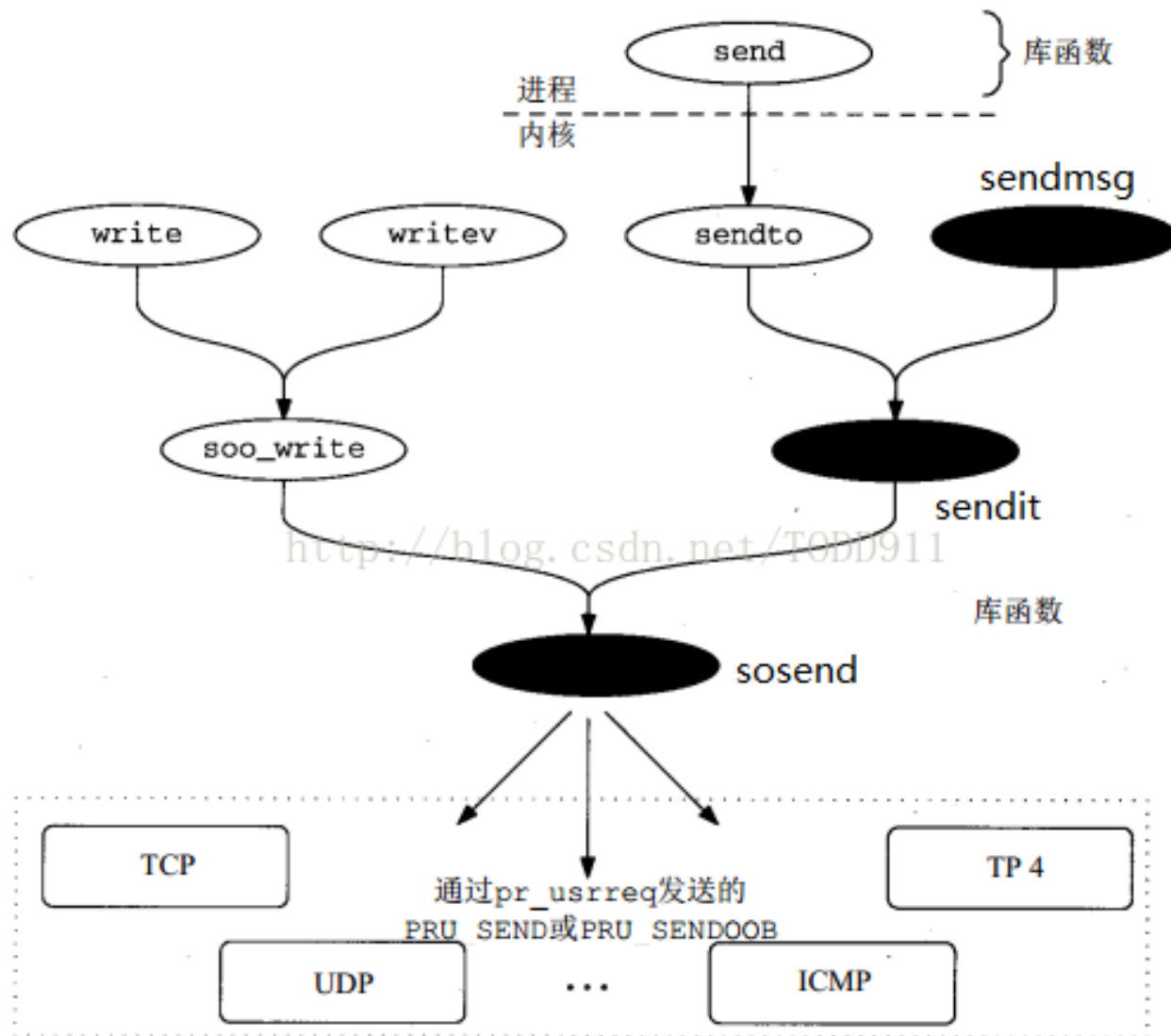
- sockfd: 套接字描述符，指明正在监听的套接字
- addr: 提出连接请求的主机地址
- addrlen: 地址长度

❑ 例：`new_sockfd = accept(sockfd, (struct sockaddr *)&address, sizeof(address));`

## 7.2.9 套接字基础—套接字API中的主要系统调用

### □ read和write

- 在Linux中，可以代替recv和send，因为都调用内核的sosend实现。
- sosend的功能是将进程来的数据复制到内核，并将数据传递给与插口相关的协议



## 套接字 API小结

(需要掌握)

函数名	含义
socket	创建用于网络通信的描述符
connect	连接远程对等实体（客户）
send (write)	通过 TCP 连接外发数据（outgoing data）
recv (read)	从 TCP 连接中获得传入数据（incoming data）
close	终止通信并释放描述符
bind	将本地 IP 地址和协议端口号绑定到套接字上
listen	将套接字置于被动模式，并设置在系统中排队的 TCP 传入连接的个数（服务器）
accept	接收下一个传入连接（服务器）
recv (read)	接收下一个传入的数据报
recvmsg	接收下一个传入的数据报（recv 的变形）
recvfrom	接收下一个传入的数据报并记录其源端点地址
send (write)	发送外发的数据报
sendmsg	发送外发的数据报（send 的变形）
sendto	发送外发的数据报，往往是到预先记录下的端点地址
shutdown	在一个或两个方向上终止 TCP 连接
getpeername	在连接到达后，从套接字中获得远程机器的端点地址
getsockopt	获得套接字的当前选项
setsockopt	改变套接字的当前选项

图 5.3 套接字函数及其含义的总结。read 和 write 与 recv 和 send 是等价的

## 7.2.9 套接字基础—用于整数转换的实用例程

- ❑ 网络字节顺序(network byte order): 最高位字节在前
- ❑ 有些套接字例程要求参数按照网络字节顺序存储, 如sockaddr\_in;
- ❑ 需要网络字节顺序和本地主机字节顺序进行转换的函数, 坚持使用, 便于移植。
- ❑ 分为短 (short 16位)和长 (long 32位)两种
  - htons: 将一个短整数从本地字节顺序转换为网络字节顺序;
  - ntohs: 将一个短整数从网络字节顺序转换为本地字节顺序;
  - htonl: 将一个长整数从本地字节顺序转换为网络字节顺序;
  - ntohl: 将一个长整数从网络字节顺序转换为本地字节顺序。

## 7.2.9 套接字基础—地址转换函数

- 人们习惯使用 202.112.14.151 表示地址（点分十进制），但是这个本质是一个字符串而不是数值，因此在socket编程时，需要进行转换。此外还要考虑字节序的问题，为此可以使用如下函数

```
#include <arpa/inet.h>
inet_aton
inet_addr
inet_ntoa
inet_pton // 建议使用这个
inet_ntop
```

仅在处理网络参数时使用，比如IP地址，端口等。而在用I/O函数接收发送数据时，不用考虑字节序问题，OS自动处理。

## 7.2.9 套接字基础—地址转换函数

- `int inet_aton(const char *cp, struct in_addr *inp)`  
返回值：1-串有效，0-串有错
  - `inet_aton`函数将`cp`所指的字符串转换成32位的**网络字节序二进制**，并通过指针`inp`来存储。这个函数需要对字符串所指的地址进行有效性验证。但如果`cp`为空，函数仍然成功，但不存储任何结果。
- `in_addr_t inet_addr(const char *cp)`  
返回值：若成功，返回32位二进制的网络字节序地址，若有错，则返回`INADDR_NONE`
  - `inet_addr`进行相同的转换，但不进行有效性验证，也就是说，所有2<sup>32</sup>种可能的二进制值对`inet_addr`函数都是有效的。

## 7.2.9 套接字基础—地址转换函数

●char \*inet\_ntoa(struct in\_addr in)

返回：指向点分十进制数串的指针

函数inet\_ntoa将32位的网络字节序二进制IPv4地址转换成相应的点分十进制数串。但由于返回值所指向的串留在静态内存中，这意味着函数是不可重入的。需要注意的是这个函数是以结构为参数，而不是指针。

上述三个地址转换函数都只能处理IPv4协议，而不能处理IPv6地址。所以建议使用以下两个函数。

## 7.2.9 套接字基础—地址转换函数

- `int inet_pton(int family, const char *src, void *dst)`

返回：1-成功，0-输入无效，-1-出错

将src指向的**字符串**转换成**二进制地址数值**放到dst中。

- `const char *inet_ntop(int family, const void *src, char *dst, size_t cnt)`

返回：指向结果的指针—成功，NULL—出错

- 将二进制数值转换成字符串。

- ◆ family参数可以是AF\_INET,也可以是AF\_INET6。

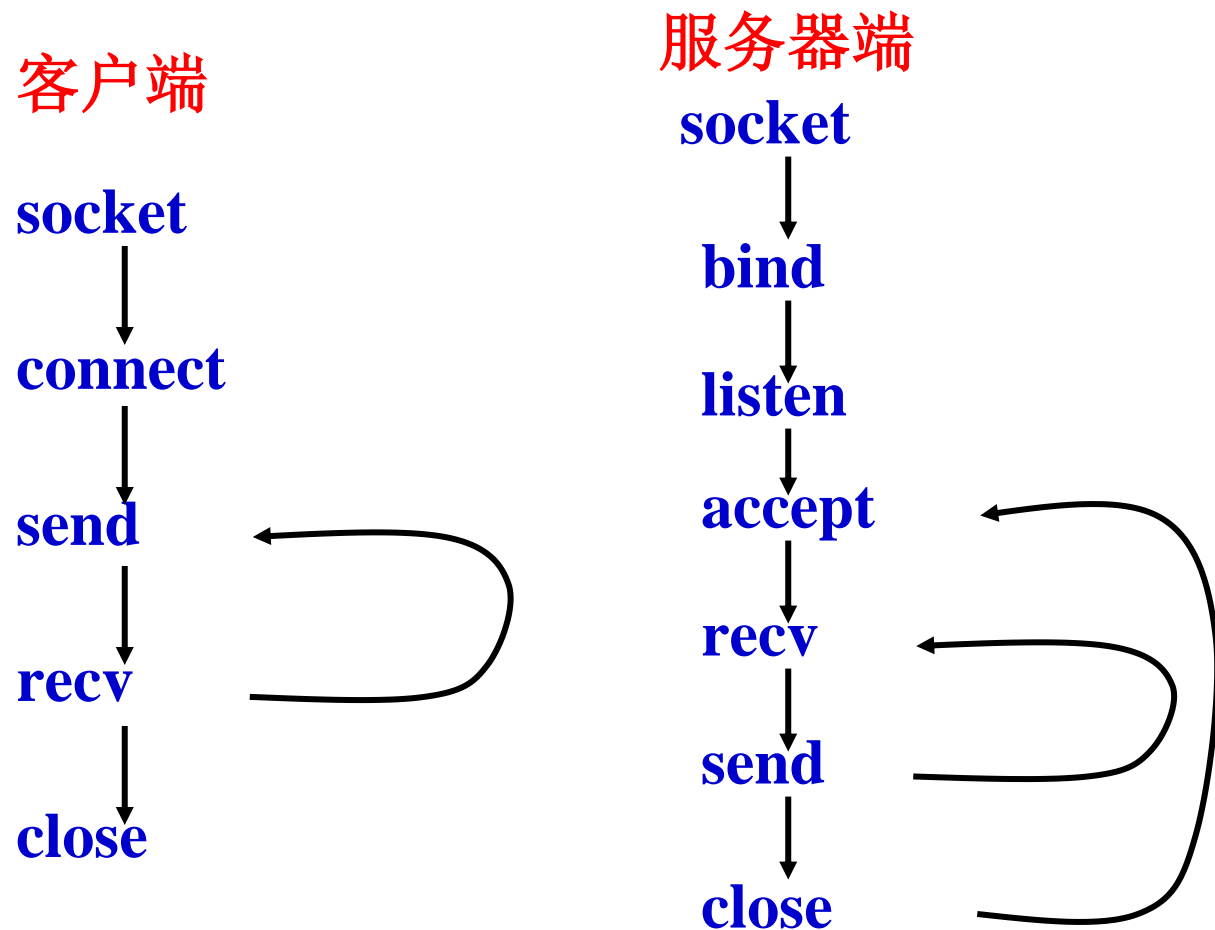
- ◆ 如果长度参数cnt太小，无法容纳表达式格式结果，则返回一个空串。另外，目标指针dst调用前必须先由调用者分配空间。



## 7.2.10 套接字基础——在程序中使用套接字调用

### □ 使用TCP的客户和服务套接字函数调用序列

图 5.4 套接字系统调用的使用序列的例子，这个序列分别由采用TCP的客户和服务端所使用。服务器一直运行。它在熟知端口上等待新连接，然后接受这个连接，与客户通信，之后便关闭这个连接



# 套接字调用参数使用的符号常量

- Unix系统提供了预定义的符号常量和数据结构来声明数据和指明参数
- 使用何种服务：
  - SOCK\_DGRAM: 数据报服务，UDP协议
  - SOCK\_STREAM: 流服务，TCP协议
  - PF\_INET: 使用TCP/IP协议族——socket() ★
  - AF\_INET: 使用TCP/IP地址结构
- 需要include，引用出现这些定义的文本
  - #include <sys/types.h>
  - #include <sys/socket.h>

## 7.2 套接字基础小结

- ❑ 套接字已经成为一种事实上的标准
- ❑ socket函数使用PF\_INET说明使用TCP/IP
- ❑ 其它的系统调用的使用方法
  - bind, listen, connect, accept, read, write, close
- ❑ 协议族的地址表示方式
  - AF\_INET指明含有一个IP地址和端口号的端点地址
  - TCP/IP是用于定义的结构sockaddr\_in
- ❑ 一些预定义的结构和常量需要include引用

## 7.3 单进程(循环)服务器/客户端程序设计

### □ 应用示例:Echo

- 客户端从键盘读取一行字符串(数据)，发送数据到服务器。
- 服务器收到数据。
- 服务器发送收到的数据给客户端。
- 客户端接收数据，在显示器上显示这行字符。

## 7.3.1 四种基本类型的服务器

- ❑ 循环的或者并发的
- ❑ 使用面向连接的或者无连接的传输

循环的无连接 (单进程)	循环的面向连接 (单进程)
并发的无连接 (多进程)	并发的面向连接 (多进程)

## 7.3.1 服务器类型

### ❑ 循环、无连接服务器

- 请求处理少量，无状态的，最常见的无连接服务器形式

### ❑ 循环的、面向连接服务器

- 要求可靠传输的，对请求要求处理少的服务，较常见

### ❑ 并发的、无连接的服务器

- 不常见，为每个请求创建一个新线程或进程

### ❑ 并发的、面向连接的服务器

- 最常见的形式。提供可靠传输，以及并发处理多个请求的能力
- 多进程可以是多个独立的程序
- 多线程（进程）或者单线程方式

## 7.3.2 简单TCP循环服务器Socket编程基本步骤

- **服务器端**

- 创建套接字
- 绑定套接字
- 设置套接字为监听模式，进入被动接受连接状态
- 接受请求，建立连接
- 读写数据
- 终止连接

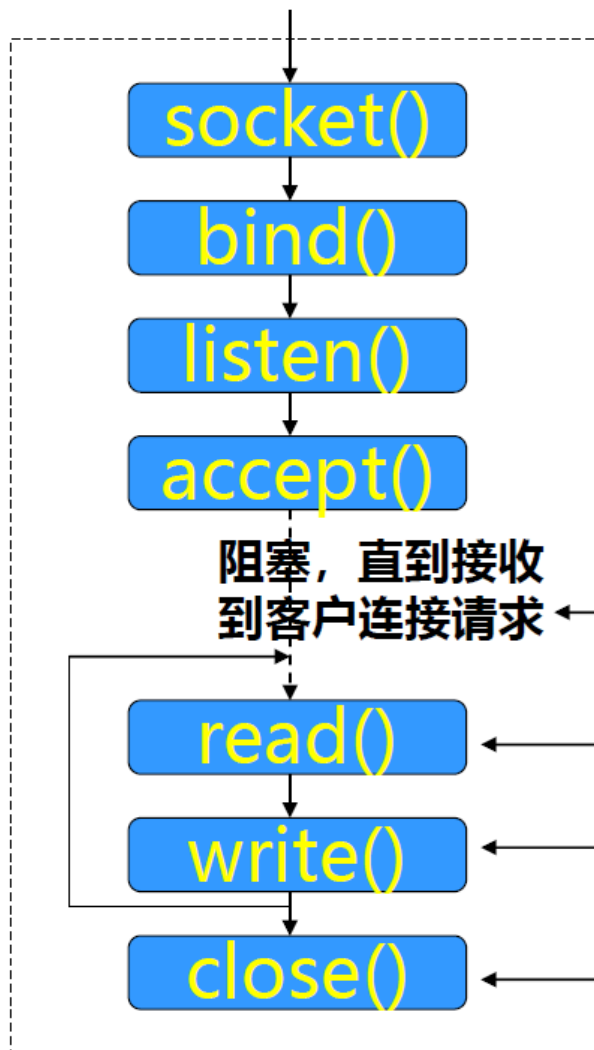
- **客户端：**

- 创建套接字
- 与远程服务器建立连接
- 读写数据
- 终止连接

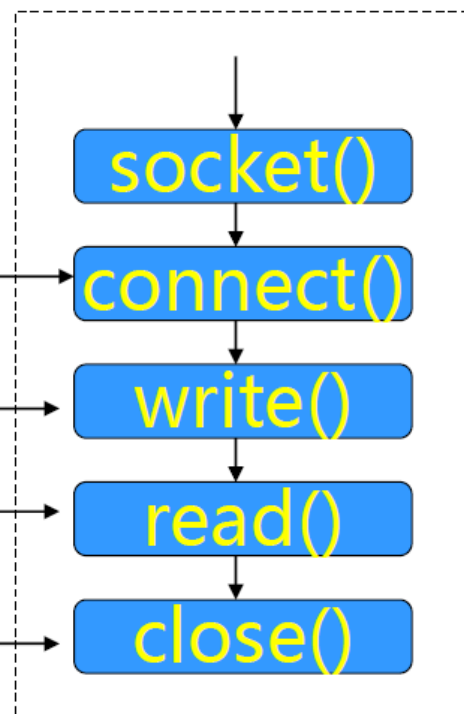
# 简单TCP循环服务器Socket编程基本步骤 ★

参考例子  
TCPEcho.c  
TCPEchod.c

TCP  
服务器端



TCP  
客户端





## 7.3.2 在客户端程序中指明服务器的地址

- ❑ 使用sockaddr\_in结构指明服务器的地址
  - 需要二进制表示的 32bit IP地址
- ❑ 套接字对地址转换的支持（两个API）
  - inet\_addr: IP地址点分十进制到二进制的转换
    - 接受一个点分十进制表示的字符串地址，返回一个等价的二进制地址
  - gethostbyname: 主机域名到二进制的转换（不要求）
    - 接受一个机器域名字符串，返回一个hostent结构，内含一个二进制表示的主机IP地址

# TCP客户算法-面向连接的客户

- ❑ 找到期望与之通信的服务器IP地址和协议端口号
- ❑ 分配套接字socket()
- ❑ 指明此连接需要在本地机器中的、任意的、未使用的协议端口，并允许TCP选择一个这样的端口
- ❑ 将这个套接字连接到服务器connect()
- ❑ 使用应用级协议与服务器通信send()/recv()
- ❑ 关闭连接close()

# TCP客户端：分配套接字

- ❑ 使用socket函数
- ❑ 将协议和服务分别说明为PF\_INET和SOCK\_STREAM
- ❑ include语句包含一些定义常量的文件
- ❑ 对于TCP/IP，第三个参数没有用。

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

#include <sys/socket.h>

int    s;  /* socket descriptor */

s = socket ( PF_INET, SOCK_STREAM, 0);
```

# TCP客户端：选择本地协议端口号

- ❑ 服务器运行于周知端口上，客户不是。
- ❑ 客户使用端口的规则：
  - 该端口不与该机器其他进程使用端口冲突
  - 该端口没有被分配给某个熟知服务
- ❑ 客户允许TCP自动选择本地端口
  - connect调用的一个效果就是所选择的本地端口能够满足上述准则。

# TCP客户端：选择本地IP地址的基本问题

- 对于只挂在一个网络上的主机是简单的
- 正确的选择依赖于选路信息，但应用程序很少使用选路信息，实际中存在的问题：
  - 一个主机可能具有多个IP地址
  - 如果应用程序随机选择一个IP地址，可能选择了一个与IP地址的接口并不匹配的地址。
  - 可能能够正确的工作。但是网络管理会困难和混乱，可靠性降低。
- 一般本地地址字段不填，允许客户自动选取本地IP地址

# TCP客户端：将TCP套接字连接到服务器

## ❑ connect函数：允许TCP套接字发起连接

- 强迫执行下层的三次握手
- 超时或者建立连接后返回
- 三个参数：
  - `retcode = connect(s, remaddr, remaddrlen);`
  - `s`: 套接字的描述符
  - `remaddr`: 一个`sockaddr_in`类型结构的地址
  - `remaddrlen`: 第二个参数的长度

## ❑ connect的四项任务

- 对指明的套接字进行检测：有效，没有连接
- 将第二个参数给出的端点地址填入套接字中
- 为此套接字选择一个本地端点地址
- 发起一个TCP连接，并返回一个值

# TCP客户端：使用TCP和服务端通信

- ❑ 客户发送请求，等待响应

- ❑ 发送请求数据报：send;

- ❑ 等待响应数据报：recv;

```
send(s, req, strlen(req), 0);
```

```
while ((n = recv(s, bptr, buflen, 0)) > 0)
```

```
{
```

```
    bptr +=n;
```

```
    buflen -=n;
```

```
}
```

- ❑ TCP不保持记录的边界，面向流的概念

# TCP客户端：关闭TCP连接

结合TCP客户端代码★★★  
TCPEcho.c来给同学们讲解

□ close：从容关闭连接释放该套接字

○ 常常需要在客户服务器之间协调关闭事宜

- 服务器不能关闭连接，不知客户请求是否完成
- 客户不知道服务器发出的数据是否全部到达

□ 允许应用程序在一个方向关闭TCP连接

○ shutdown(s, direction);

- direction: 0不允许输入; 1不允许输出; 2双向关闭

○ 部分关闭可以让服务器发送完最后一个响应后，关闭连接。



## 7.3.4 TCP的服务器端设计

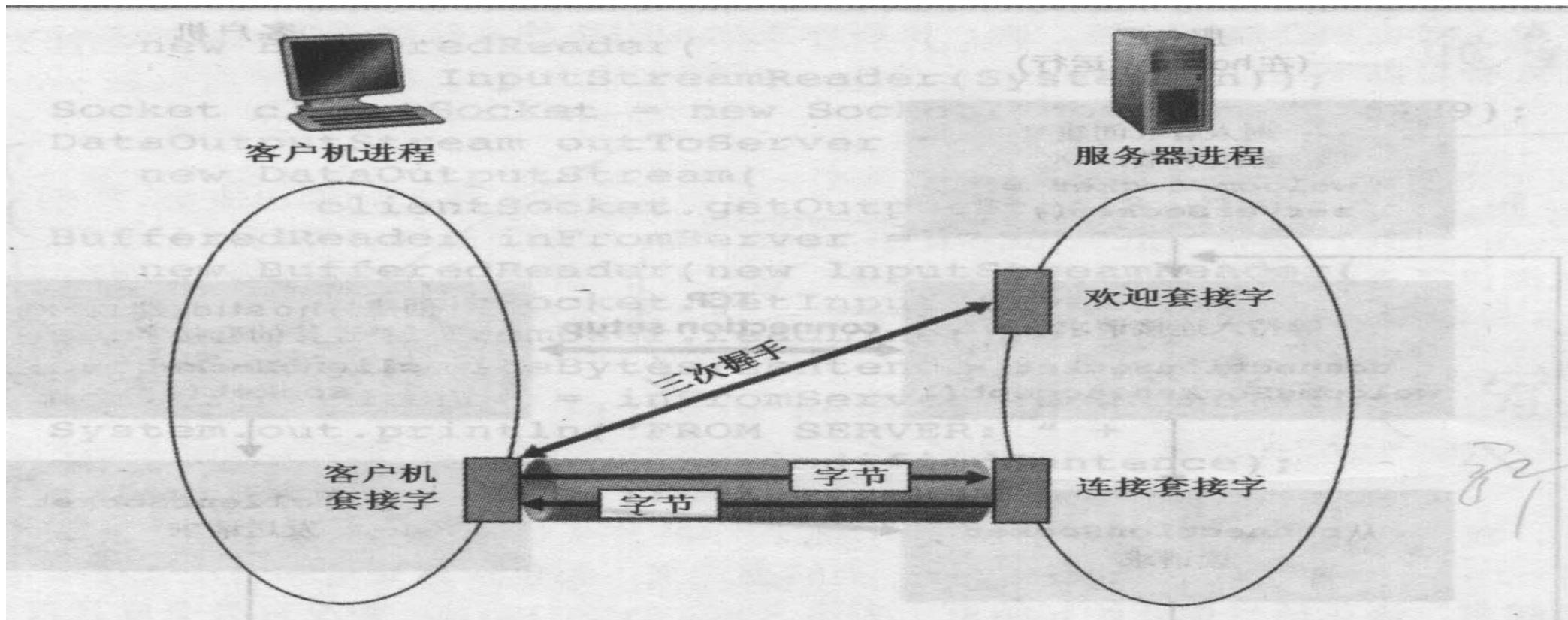
### ❑ 面向连接的服务的优点：

- 易于编程
  - 自动处理分组丢失，分组失序
  - 自动验证数据差错，处理连接状态

### ❑ 面向连接的服务的缺点：

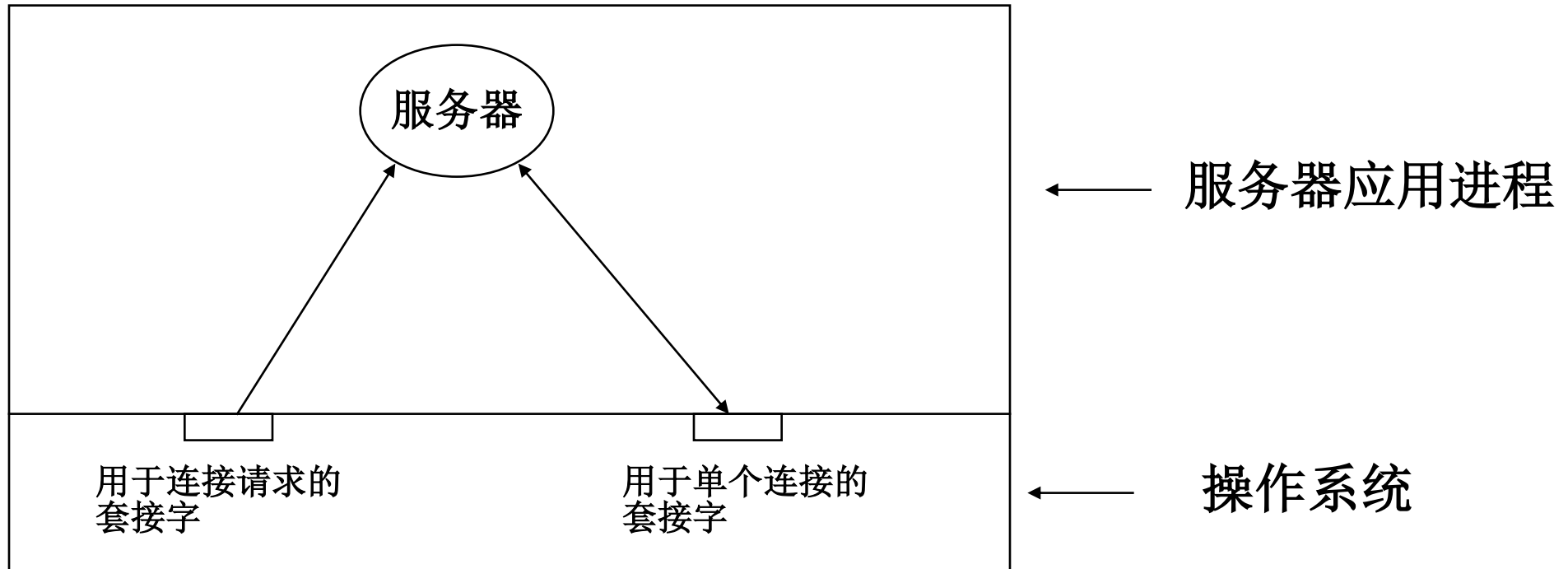
- **对每个连接都有一个单独的套接字**，耗费更多的资源
- 在空闲的连接上不发送任何分组
- 始终运行的服务器会因为客户的崩溃，导致无用套接字的过多而耗尽资源

# 采用TCP的单进程(循环)服务器设计



# 采用TCP的单进程(循环)服务器设计

- 使用一个进程
- 使用两个套接字
  - 一个套接字处理请求
  - 另外一个套接字处理和客户的通信（临时的）



# 采用TCP的单进程(循环)服务器算法

## □ 基于TCP的循环服务器算法

- 1、创建套接字并将其绑定到它所提供服务的熟知端口上；`socket+bind`
- 2、将该端口设置为被动模式，使其准备为服务器所用；`listen`
- 3、从该套接字上接收下一个连接请求，获得该连接的新的套接字；  
`ssock=accept(s,)`
- 4、重复地读取来自客户的请求，构造响应，按照应用协议向客户发回响应；  
`send/rcv(ssock)`
- 5、当某个特定客户完成交互时，关闭连接，并返回步骤3以接受新的连。  
`close(ssock)`

# 用INADDR\_ANY设置通配地址

- ❑ 服务器需要创建欢迎套接字并将其绑定到所熟知的端口上
  - bind为某个套接字指明某个端点，使用结构sockaddr\_in，该结构含有IP地址和端口号
  - 对于多接口主机使用**INADDR\_ANY**指明了一个通配地址，让该主机的任何一个IP地址都匹配。

```
msock=socket(PF_INET, SOCK_STREAM, 0)); //创建TCP套接字
servaddr.sin_family=AF_INET; //设置IPv4地址族
servaddr.sin_addr.s_addr=INADDR_ANY; //设置通配地址
servaddr.sin_port=htons(8080);
bind(msock, (struct sockaddr*)&servaddr, sizeof(servaddr));
```

# 将套接字置于被动模式

## □ 调用listen：将套接字置于被动模式

```
listen(msock, QLEN);
```

- 一个参数指明套接字内部的请求队列长度
- 请求队列保存一组TCP传入连接请求(来自客户)，都向这个服务器请求一个连接

## □ 接收连接并使用这些连接

- 调用accept：获得下一个传入连接请求
- 返回新的连接的套接字的描述符
- 服务器接收连接，使用read获得来自客户的应用协议，使用write发回应答
- 服务器结束连接，使用close释放套接字

```
alen = sizeof(fsin);  
ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
```

```
close(ssock); //关闭套接字
```

# TCP ECHO服务的实例

循环、面向连接的服务器(参考代码讲解)★

结合TCP服务器代码  
TCPEchod.c来给同学们讲解

## 7.3.5 无连接、循环服务器的算法

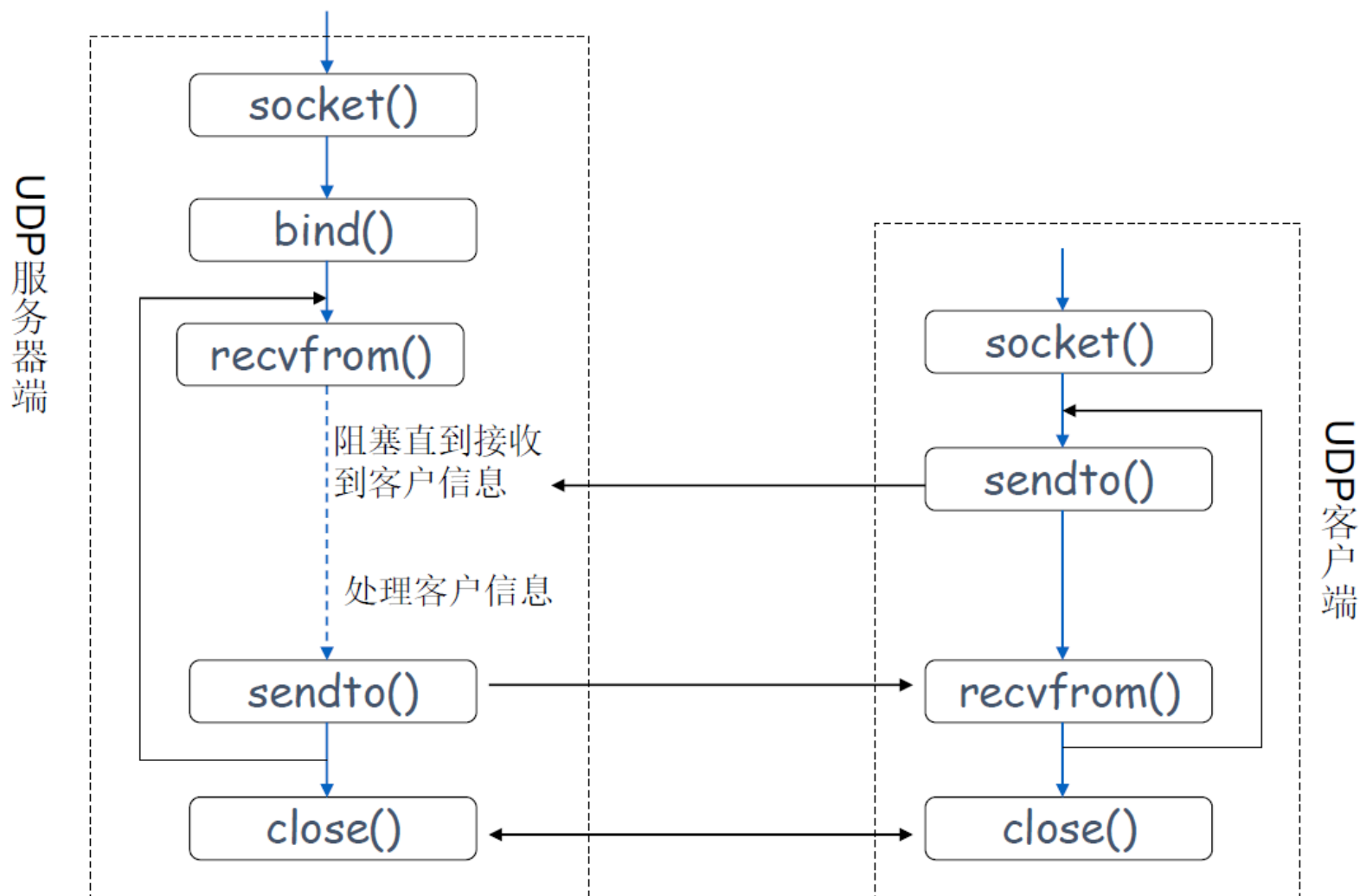
- ❑ 循环服务器的设计、编程、排错、修改很容易
  - 往往使用无连接的协议。
- ❑ 循环服务器对于小的处理时间的服务工作很好。
- ❑ 无连接服务器算法如下：
  - 1、创建套接字并将其绑定到所提供服务的熟知端口上；
  - 2、重复读取来自客户的请求，构造响应，按照应用协议向客户发回响应。



# 简单UDP循环服务器Socket编程基本步骤

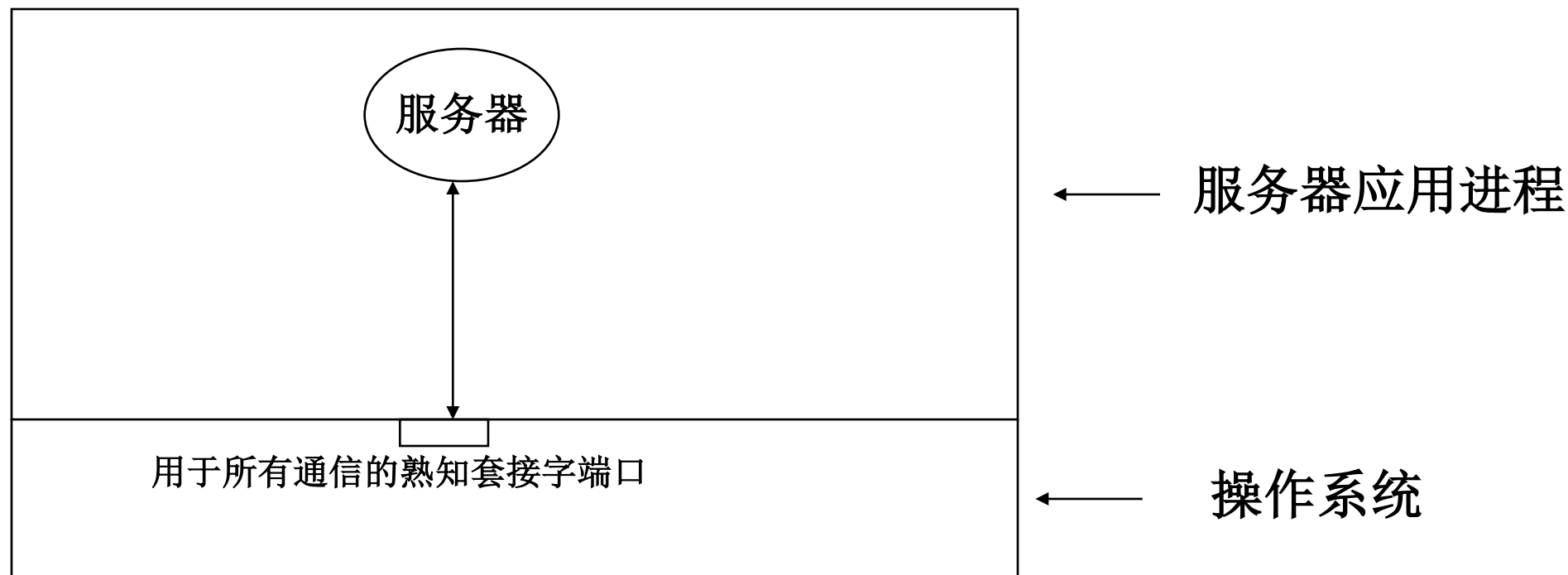
- 实现UDP套接字基本步骤分为服务器端和客户端两部分
- 服务器端
  - ① 建立UDP套接字;
  - ② 绑定套接字到特定地址;
  - ③ 等待并接收客户端信息;
  - ④ 处理客户端请求;
  - ⑤ 发送信息回客户端;
  - ⑥ 关闭套接字;
- 客户端步骤
  - ① 建立UDP套接字;
  - ② 发送信息给服务器;
  - ③ 接收来自服务器的信息;
  - ④ 关闭套接字

# 简单UDP循环服务器Socket编程基本步骤



# 循环(单进程)、无连接(UDP)服务器的进程结构

- 循环、无连接服务器的进程结构
  - 只需要一个执行进程



## 7.3.6 UDP客户端

- ❑ 找到期望与之通信的服务器IP地址和协议端口号
- ❑ 分配套接字socket()
- ❑ 指明这种通信需要本地机器中的、任意的、未使用的协议端口，并允许UDP选择一个这样的端口connect()
- ❑ 指明报文所要发往的服务器sendto()
- ❑ 使用应用级协议与服务器通信sendto()/recvfrom()
- ❑ 关闭连接close()

# UDP客户端

## □ 非连接的UDP通信

- 在每次发送报文的时候指明远程目的地
- 使用灵活，便于同不同的服务器通信

## □ 对于非连接的UDP套接字

- sendto: 发送报文，含有地址信息
- recvfrom: 接收一个含有源地址的数据报

# UDP客户端

无连接服务器使用sendto：指明了发送的数据报和它将去的地址

套接字 指向要发 要发送数 标志位 接收主机 地址长度  
描述符 送的数据 据的长度 设为0 的地址

```
retcode = sendto(s, msg, len, flags, toaddr, toaddrlen);
```

服务器从收到的报文中的**源地址获得应答的地址**。调用recvfrom得到数据和对方的地址

套接字 指向存放 接收数据 标志位 发送主机 地址长度  
描述符 接收数据 缓冲区的 设为0 的地址

```
retcode = recvfrom(s, buf, len, flags, from, fromlen);
```

由收到的数据报文的地址信息来设置，写代码的时候不需要设置

# 关闭UDP套接字和UDP特点

- ❑ close: 关闭套接字，释放与之关联的资源
  - 拒绝以后到达的报文
  - 没有通知远程端点
- ❑ shutdown: 在某个方向上终止进一步传输
  - 不向另外一方发送任何报文，只是在本地套接字标明不期望在指定的方向传输数据
  - 客户关闭输出以后，服务器并不知道
- ❑ UDP提供的是不可靠的交互
  - 必须自己设计协议实现可靠性

## 7.3.7 UDP ECHO服务

- ❑ ECHO服务器返回从客户收到的所有数据
- ❑ 用户网络管理员测试可达性，调试协议软件，识别选路问题等
- ❑ **UDP ECHO服务**：接收整个数据报，根据数据报指明的端口号和地址，返回整个数据报

参考提供的UDP ECHO的客户端和服务端代码（★）



## 7.4 并发(多进程)服务器程序

- ❑ 循环服务器：一个时刻只处理一个请求
- ❑ 并发服务器：一个时刻可以处理多请求
  - 多数只提供表面并发：执行多个线程，每个线程处理一个请求
  - 使用单线程的可能性：计算量小，主要是异步I/O，便于同时使用多个通信信道
  - 并发处理多个请求，而不是指下层是否使用了多个并发线程
- ❑ 循环服务器容易构建，但是性能差；并发服务器难以构建和设计，但是性能好

## 7.4.1 并发服务器的算法

- 给多个客户提供快速响应时间需要使用并发服务器
  - 有相当的I/O时间的响应
    - 可以部分重叠地使用处理器和外设
  - 各个请求所要求的处理时间变化很大
    - 时间分片允许单个处理器处理那些只要求少量处理的请求尽快完成
  - 服务器运行在具有多个处理器的计算机上
    - 不同的处理器处理不同的请求

## 7.4.1 主线程和从线程

- 尽管可以使用一个单线程实现并发服务器，但是大多数使用多线程：
  - 主线程最先开始执行在**熟知的端口上打开一个套接字**，等待一个请求，并为每个请求创建一个从线程（可能在一个新进程中）
  - **主线程不与客户直接通信，每个从线程处理一个客户的通信。**
  - 从线程构成响应并发送给客户后，这个从线程便退出

## 7.4.2 并发的无连接的服务器的算法

### □ 最简单的算法：

- 主1、创建套接字并将其绑定到所提供服务的熟知地址上。让该套接字保持为未连接的
- 主2、反复调用recvfrom接收来自客户的下一个请求，创建一个新的从线程来处理响应
  - 从1、从来自主进程的特定请求以及到该套接字的访问开始
  - 从2、根据应用协议构造应答，并用sendto将该应答发回给客户
  - 从3、退出（即：从线程处理完一个请求后就终止）

### □ 由于创建进程或者线程是昂贵的，因此只有很少的无连接服务器采用并发实现

## 7.4.3 并发的面向连接服务器算法

□ 面向连接的服务器在多个连接之间实现并发（不是在各个请求之间）

主1、创建套接字并将其绑定到所提供服务的熟知地址上。让该套接字保持为面向连接

主2、将该端口设置为被动模式

主3、反复调用`accept`以便接收来自客户的下一个连接请求，并创建新的从线程或者进程来处理响应

从1、由主线程传递来的连接请求开始

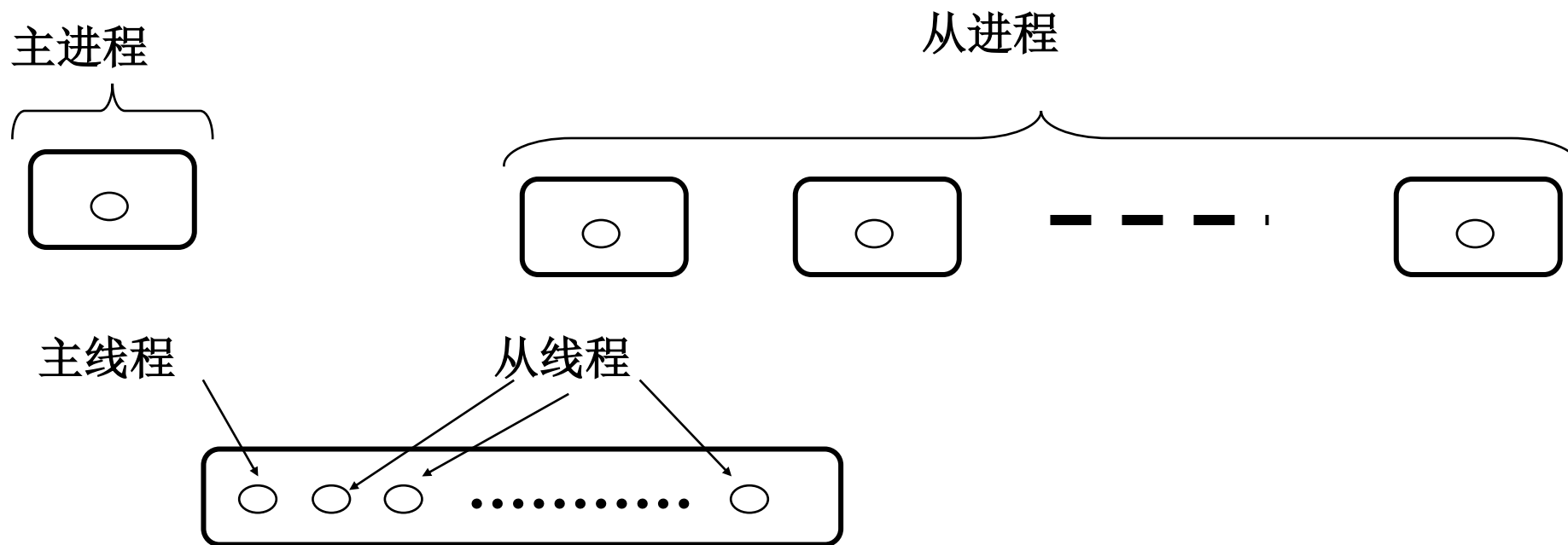
从2、用该连接与客户进行交互；读取请求并发送回响应

从3、关闭连接并退出

# 服务器并发性的实现

## □ 两种形式的并发性：进程和线程

- 服务器创建多个进程，每个进程都有一个执行线程
- 服务器在一个进程中创建多个执行线程



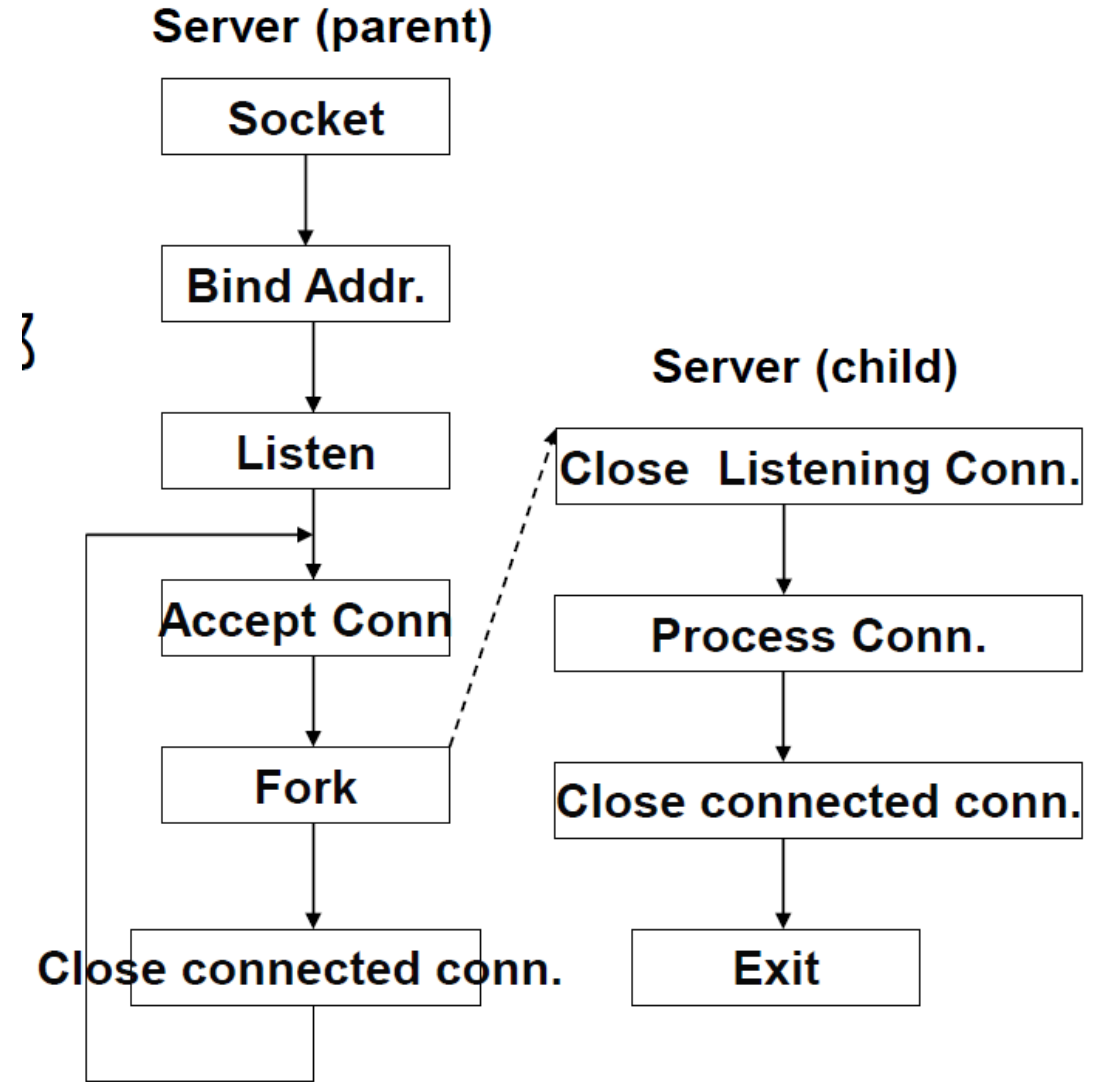
# 把单独的进程作为进程使用

- ❑ 并发服务器为每个连接创建一个新从线程
  - 对于单线程的进程实现，采用fork实现
  - 程序中包含主进程和从进程的全部代码
- ❑ 从进程执行一个单独编写和编译的程序也许更加方便
  - Linux系统支持
  - 调用fork后再调用execve/exec1()

# 并发的、面向连接的服务器

## □ 并发的、面向连接的服务器

- 主服务器进程在机器启动的时候自动一直运行，对每个客户的新连接创建一个新的从线程/进程进行处理；
- 讨论基于多进程(每个进程包含一个线程)设计的并发服务器。

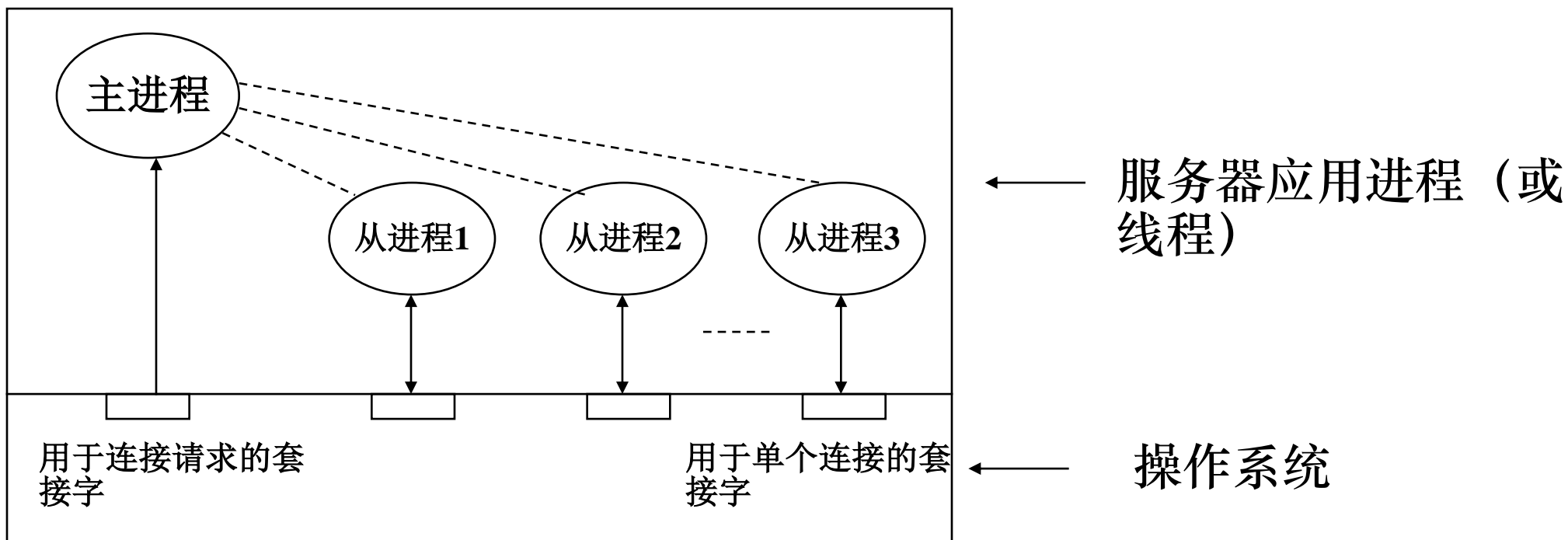


TCP Concurrent Server



# 多进程并发服务器的结构

- ❑ 服务器包括一个主进程，以及零个或者多个从进程。每个进程一个线程
- ❑ 主进程使用accept阻塞调用，节约CPU资源，连接到来的时候，accept马上返回。



# 代码示例：并发TCP ECHO

- 功能：客户打开到某个服务器的连接，然后在该连接上发送数据，并读取从服务器返回的数据并回显，服务器响应每个客户，接受来自每个客户的连接，读取来自客户的数据，并原样返回给客户。
  - 服务器在发送响应前并非读取全部输入，只是交替读写
  - 服务器在遇到文件结束的条件后，关闭连接

# 并发ECHO服务器举例

- ❑ #include 语句
- ❑ 变量定义，宏定义，函数声明
- ❑ 主函数
  - 参数处理
  - 建立被动套接字
  - 循环等待连接，如果有新连接，则fork一个新的线程，调用TCPEchod进行处理
- ❑ TCPEchod函数
  - 处理echo服务

```
switch (fork()) {  
    case 0:                /* child */  
        (void) close(msock);  
        exit(TCPEchod(ssock));  
    default:                /* parent */  
        (void) close(ssock);  
        break;  
    case -1:  
        errexit("fork: %s\n", strerror(errno));  
}
```

# 清除游离进程

- 使用fork的服务器动态生成进程，可能导致不完全的进程终止
  - Linux在一个子进程退出的时候，会给父进程一个信号（signal）
  - 正在退出的进程保持僵死状态，直到父进程执行wait3系统调用为止
  - Signal(SIGCHLD, reaper)主服务器进程收到子进程退出信号的时候，执行函数reaper
  - 函数reaper调用函数wait3完成子进程的终止并退出
    - 参数WNOHANG指明wait3不要为了进程退出而阻塞等待

# 补充—僵死进程

- 僵死(zombie)进程：父进程创建的子进程已完成任务，交回所占用的大部分资源，但是该进程还未从系统中删除，等待父进程对其进行回收（获取终止子进程的有关信息，释放它仍占用的资源）。如果父进程不对其进行专门的回收，这些子进程就成为僵死进程。
  - 如果子进程完全消失了，父进程在最终准备好检查子进程是否终止时，是无法获取它的最终状态的。
  - 内核为每个终止子进程保存了一定量的信息，所以当终止进程的父进程调用wait或waitpid时，可以得到这些信息。得到的信息包括：进程ID、该进程的终止状态，以及该进程使用的CPU时间总量。
  - 通过wait或waitpid函数，内核可以释放终止进程所使用的所有存储区，关闭其所有打开文件。

# 补充—僵死进程

- ❑ 对终止子进程进行资源回收，可以通过waitpid(wait)函数与SIGCHLD信号一起完成。
  - 当子进程终止，内核向父进程发送**SIGCHLD**信号
  - 父进程可以在SIGCHLD的信号处理函数中调用wait/waitpid处理子进程的遗留状态

```
#include<sys/wait.h>
pid_t wait(int* statloc);
pid_t waitpid(pid_t pid, int* statloc, int options);
```

如果子进程终止，返回值都是进程号PID；  
如果函数参数中设置WNOHANG，且子进程没有终止，返回值是0；  
如果函数调用出错，返回值是-1

# 补充—僵死进程

## □ wait函数与waitpid函数的区别

- 当调用wait函数时，如果没有子进程终止，则调用者会一直阻塞，直到有个子进程终止，则wait函数立即收回终止子进程的资源，并返回。
- 当调用waitpid函数时，如果没有子进程终止，也可以通过设置函数的参数选项，使调用者不阻塞，当有子进程终止时，waitpid函数返回终止子进程的进程ID，并将该子进程的终止状态存放在由statloc指向的存储单元中。

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

终止子进程  
的进程号

终止子进程的  
信息存放地址

设置选项

参数pid	说明
pid==-1	等待任何1个子进程退出，没有任何限制，此时waitpid和wait的作用一模一样。
pid>0	只等待进程ID等于pid的子进程，不管其它已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid就会一直等下去。
pid==0	等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，waitpid不会对它做任何理睬。
pid<-1	等待一个指定进程组中的任何子进程，这个进程组的ID等于pid的绝对值。

options参数使我们能进一步控制waitpid的操作。此参数或者是0，或者是以下**常量按位“或”运算的结果**。

- **WNOHANG**：若由pid指定的子进程未发生状态改变(没有结束)，则waitpid()不阻塞，立即返回0；（我们实验中用到的是这个常量）
- WUNTRACED：返回终止子进程信息和因信号停止的子进程信息；
- WCONTINUED：返回收到SIGCONT信号而恢复执行的已停止子进程状态信息



# 补充—僵死进程

## □ 函数wait和waitpid

- 当一个进程正常或异常终止时，内核就向其父进程发送**SIGCHLD**信号。因为子进程终止是个异步事件(可以在父进程运行的任何时候发生)，所以这种信号也是内核向父进程发的异步通知。**父进程可以选择忽略该信号，或者提供一个该信号发生时即被调用执行的函数（信号处理程序，我们网络编程中就是采用这种方式）。**
- 对于SIGCHLD信号，系统的默认操作是忽略它。因此为了捕捉SIGCHLD信号，我们需要设置信号，并定义处理函数；当捕捉到SIGCHLD信号时，就执行处理函数。

在父进程中设置信号

`sigaction()`函数

`signal(SIGCHLD, sig_child);`

signal函数的使用方法简单，但并不属于 POSIX 标准，在各类 UNIX 平台上的实现不尽相同，因此其用途受一定限制。而 POSIX 标准定义的信号处理接口是 `sigaction` 函数

## 补充—SIGCHLD信号

```
//定义SIGCHLD信号的相关信息
struct sigaction sigact_chld, old_sigact_chld;
sigact_chld.sa_handler = &sig_chld;
sigemptyset(&sigact_chld.sa_mask);
sigact_chld.sa_flags = 0;
//设置受影响的慢系统调用重启
sigact_chld.sa_flags |= SA_RESTART;
//sig_chld()函数是SIGCHLD信号的执行函数
sigaction(SIGCHLD, &sigact_chld, &old_sigact_chld);
```

# 补充—僵死进程

□ 调用wait和waitpid函数后，发生的情况：

- 如果所有子进程都还在运行，则阻塞；
- 如果1个子进程已终止，正等待父进程获取其终止状态，则取得该子进程的终止状态立即返回；
- 如果它没有任何子进程，则立即出错返回；
- 如果设置为WNOHANG选项，若子进程没结束，waitpid函数返回0，不阻塞。

最后给学生展示  
单进程、多进程  
处理相关代码

```
void sig_child(int sign) {  
    pid_t pid;  
    int stat;  
    while ((pid = waitpid(-1,&stat,WNOHANG)) > 0) {  
        printf("child %d terminated.\n", pid);  
        return;  
    }  
}
```

## 7.4 基于多进程实现并发服务器小结

- ❑ 面向连接的并发技术
- ❑ 使用多进程的方式实现：fork
  - 主进程的线程永远不会和任何客户打交道，只接受连接，创建一个从进程处理各个连接
- ❑ 从进程从主进程调用fork后立即执行
  - 主进程关闭新连接所用的描述符的副本
  - 从进程关闭主描述符的副本

process-echod.c (★) 多进程TCP echo服务器

## 与大家共勉

莫聽穿林打葉聲  
何妨吟嘯且徐行  
竹杖芒鞋輕勝馬  
誰怕一簑煙雨任  
平生料峭春風吹  
酒醒微冷山頭斜  
照却相迎回首向  
來蕭瑟處歸去也  
無風雨也無晴

三月七日沙湖道中遇雨雨具先亡同  
行皆狼狽餘強不覺已而遂晴故作此

歲在辛卯秋月馬立林

