

网络程序设计

北京邮电大学计算机学院 蒋砚军
jiangyanjun0718@bupt.edu.cn

Socket概述

Socket

■ 协议栈实现

- ◆ 传输层以上由用户态应用程序实现
- ◆ 传输层和网络互联层协议在内核中实现（路由协议由用户态进程实现）
- ◆ 第一第二层一般由硬件实现

■ UNIX提供给应用程序使用网络功能的方法

- ◆ BSD将设备和通信管道组织成文件方式，创建方式不同，访问方法相同
 - 终端设备
 - 管道
 - 通信服务Socket
- ◆ AT&T UNIX的TLI编程接口

■ Socket编程接口面向网络通信，不仅仅用于TCP/IP

- ◆ 利用虚拟loopback接口(127.0.0.1)，可实现同台计算机进程间通信

TCP与UDP

■ TCP

- ◆ 面向连接
- ◆ 可靠
- ◆ 字节流传输
 - 不保证报文边界

■ UDP

- ◆ 面向数据报
- ◆ 不可靠
 - 错报, 丢报, 乱序, 流量控制
- ◆ 数据报传输
- ◆ 广播和组播

网络字节顺序

■ CPU字节顺序

- ◆ Big Endian (大尾)

 - Power PC, SPARC, Motorola

- ◆ Little Endian (小尾)

 - Intel X86

■ 网络字节顺序

- ◆ 与X86相反

■ 网络字节转换的库函数

- ◆ htonl ntohl 四字节整数(long)

- ◆ htons ntohs 两字节整数(short)

TCP客户-服务器程序

客户端程序：client.c(1)

```
#define SIZE 8192
#define PORT_NO 12345

int main(int argc, char *argv[])
{
    int sock, len;
    struct sockaddr_in name;
    unsigned char sbuf[SIZE];

    if (argc < 2) .....
    sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock < 0) .....

    name.sin_family = AF_INET;
    inet_aton(argv[1], &name.sin_addr);
    name.sin_port = htons(PORT_NO);
    if (connect(sock, (struct sockaddr *)&name, sizeof(name)) < 0) {
        perror("\nconnecting server stream socket");
        exit(1);
    }
    printf("Connected.\n");
}
```

客户端程序：client.c(2)

```
for (;;) {  
    if (fgets(sbuf, sizeof sbuf, stdin) == NULL)  
        break;  
  
    if (write(sock, sbuf, strlen(sbuf)) < 0) {  
        perror("sending stream message");  
        exit(1);  
    }  
}  
close(sock);  
printf("Connection closed.\n\n");  
exit(0);  
}
```


客户端程序

- 创建文件描述符socket
- 建立连接connect
 - ◆ 进程阻塞，等待TCP连接建立
- 端点名的概念：IP地址+端口号
 - ◆ 本地端点名
 - ◆ 远端端点名
- 发送数据
 - ◆ 发送速率大于通信速率，进程会被阻塞
- 关闭连接

服务端程序：server0.c(1)

```
#define PORT_NO 12345
#define SIZE 8192
```

```
int main(void)
{
```

```
    int admin_sock, data_sock;
    struct sockaddr_in name;
    unsigned char buf[SIZE];
    int nbyte, i;
```

```
    admin_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
    name.sin_family = AF_INET;
    name.sin_addr.s_addr = INADDR_ANY;
    name.sin_port = htons(PORT_NO);
    bind(admin_sock, (struct sockaddr*)&name, sizeof(name));
    listen(admin_sock, 5);
```

```
    data_sock = accept(admin_sock, 0, 0);
    printf("Accept connection\n");
```

服务端程序：server0.c(2)

```
for (;;) {
    nbyte = read(data_sock, buf, SIZE);
    if (nbyte == 0) {
        printf("*** Disconnected.\n");
        close(data_sock);
        exit(0);
    }
    for (i = 0; i < nbyte; i++)
        printf("%c", buf[i]);
}
```

socket系统调用

■ socket

- ◆ 创建文件描述符socket，端点名未指定

■ bind

- ◆ 设定本地端点名，也可以用在客户端程序

■ listen

- ◆ 开始监听到达的连接请求

■ accept

- ◆ 接受一个连接请求，TCP三次握手结束accept返回

■ connect

- ◆ 建立连接，设定远端端点名，TCP连接建立，函数返回

■ close

- ◆ 关闭连接，释放文件描述符

服务端程序

■ bind

- ◆ 设定本地端点名
- ◆ 也可以用在客户端程序

■ listen

- ◆ 进程不会在此被阻塞，仅仅给内核一个通知

■ accept

- ◆ 进程会在这里阻塞等待新连接到来

■ 创建新进程时的文件描述符处理

■ 问题：不能同时接纳多个连接

- ◆ 多进程并发处理
- ◆ 单进程并发处理

多进程并发处理

多进程并发：server1.c(1)

```
#define PORT_NO 12345
```

```
int main(void)
```

```
{
```

```
    int admin_sock, data_sock, pid, name_len;
```

```
    struct sockaddr_in name, peer;
```

```
    admin_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
    if (admin_sock < 0) .....
```

```
    name.sin_family = AF_INET;
```

```
    name.sin_addr.s_addr = INADDR_ANY;
```

```
    name.sin_port = htons(PORT_NO);
```

```
    if (bind(admin_sock, (struct sockaddr*)&name, sizeof(name)) < 0) ...
```

```
    listen(admin_sock, 5);
```

```
    signal(SIGCLD, SIG_IGN); /* 必须执行此操作,否则进程僵尸问题 */
```

多进程并发：server1.c(2)

```
for (;;) {
    name_len = sizeof(peer);
    data_sock = accept(admin_sock, (struct sockaddr *)&peer, &name_len);
    if (data_sock < 0) continue;
    printf("Accept connection from %s:%d\n",
        inet_ntoa(peer.sin_addr), ntohs(peer.sin_port));
    pid = fork();
    if (pid > 0) { /* parent process */
        close(data_sock); /* 必须执行此操作, 原因有2 */
    } else if (pid == 0) { /* child process */
        char fd_str[16];
        close(admin_sock);
        sprintf(fd_str, "%d", data_sock);
        execlp("./server1a", "./server1a", fd_str, NULL);
        perror("execlp");
        exit(1);
    }
}
```


多进程并发：server1a.c(1)

```
#define SIZE      8192
int main(int argc, char **argv)
{
    int i, nbyte, sock = strtol(argv[1], 0, 0);
    struct sockaddr_in peer, local;
    socklen_t name_len = sizeof(peer);
    unsigned char buf[SIZE];

    getpeername(sock, (struct sockaddr *)&peer, &name_len);
    getsockname(sock, (struct sockaddr *)&local, &name_len);

    for (;;) {
        nbyte = read(sock, buf, SIZE);
        printf("%s:%d => ", inet_ntoa(peer.sin_addr), ntohs(peer.sin_port));
        printf("%s:%d ", inet_ntoa(local.sin_addr), ntohs(local.sin_port));
        if (nbyte == 0) {
            printf("*** Disconnected.\n");
            close(sock);
            exit(0);
        }
        for (i = 0; i < nbyte; i++)
            printf("%c", buf[i]);
    }
}
```

端点名相关的系统调用

■ **getpeername**获取对方的端点名

```
getpeername(int sockfd, struct sockaddr *name, int *namelen);
```

■ **getsockname**获取本地的端点名

```
getsockname(int sockfd, struct sockaddr *name, int *namelen);
```

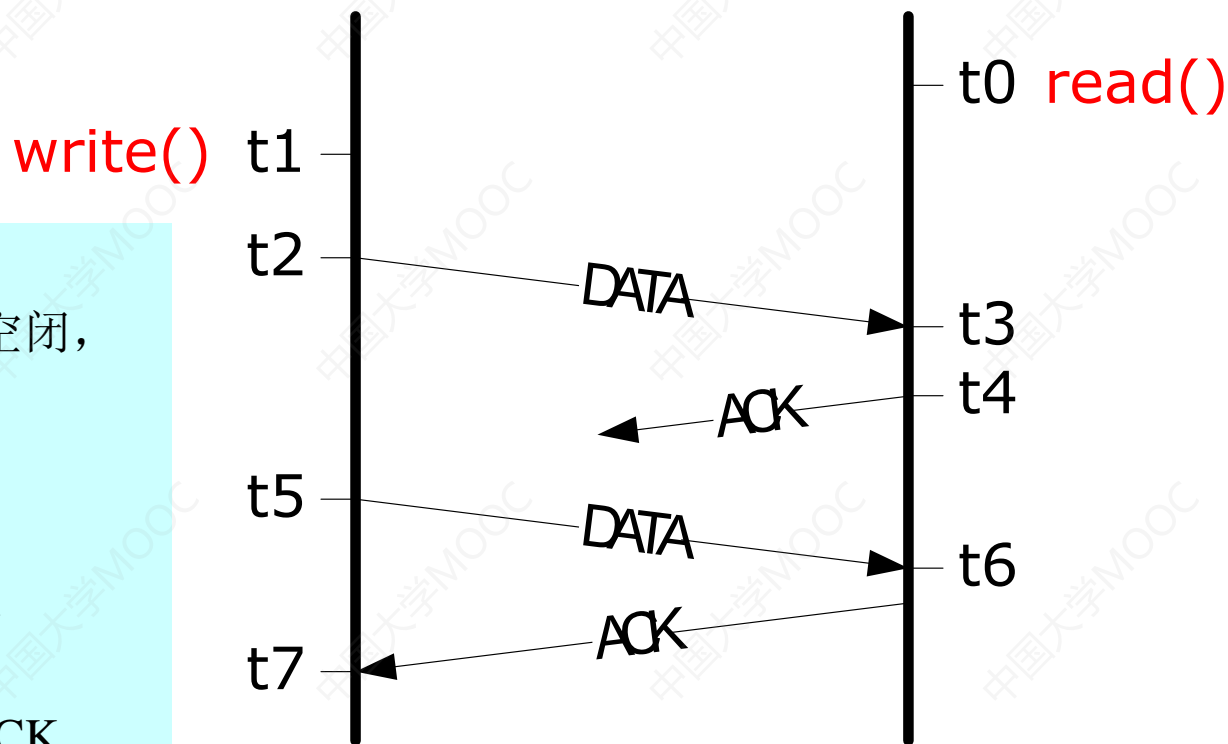
注意：namelen是传入传出型参数，函数调用之前要先给整数namelen赋值，指出name缓冲区的可用字节数；函数返回时，namelen表示实际在name处写入了多少字节有效数据

read/write系统调用的语义(1)

■ read/write与TCP通信的时序

主机A用write()通过TCP连接向B发送数据，B接收数据用read()

t0: B开始read()
t1: A调用write(), TCP发送缓冲区有空闲,
数据拷贝至发送缓冲区
t2: A将数据发往B
t3: B收到数据后, 校验和正确
t4: B向主机A发ACK, ACK途中丢失
t5: A超时自动重发数据
t6: B收到重复的数据后扔掉, 回送ACK
t7: A收到ACK, 将发送缓冲区的数据清除



read/write系统调用的语义(2)

- ◆ write在t1返回, read在t3返回

■ read/write与TCP通信故障和流控

- ◆ 流控问题

- ◆ 断线

- ◆ 对方重启动

- ◆ Keepalive(默认两小时)

- ◆ getsockopt/setsockopt可以设置保活间隔, 重传次数, 重传时间

■ “粘连问题”

■ read/write与UDP通信

- ◆ 网络故障

- ◆ 没有数据粘连

- ◆ 没有流控功能

- ◆ 不可靠

read/write的其他版本

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

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

```
int send(int sockfd, void *buf, int nbyte, int flags);
```

```
int sendto(int sockfd, void *buf, int nbyte, int flags, struct sockaddr *to, int tolen);
```

recvfrom/sendto可以指定对方的端点名，常用于UDP

Winsock只能用recv/send不可用read/write

shutdown系统调用(1)

■ `int shutdown(int sockfd, int howto);`

◆ 禁止发送或接收。socket提供全双工通信，两个方向上都可以收发数据，shutdown提供了对于一个方向的通信控制

■ 参数`howto`取值

◆ SHUT_RD: 不能再接收数据，随后read均返回0

◆ SHUT_WR: 不能再发送数据，本方向再次write会导致SIGPIPE信号

◆ SHUT_RDWR: 禁止这个`sockfd`上的任何收发

shutdown系统调用(2)

- shutdown是通用的套接字上的操作
- 执行后对通信的影响，会与具体的通信协议相关
- TCP协议
 - ◆ 允许关闭发送方向的半个连接
 - ◆ 没有一种机制让对方关闭它的发送，但TCP协议的流量控制机制，可以通知对方自己的接收窗口为0，对方的write会继续，并将数据堆积在发送缓冲区
- UDP协议
 - ◆ UDP关闭接收方向内核仅记下一个标记，不再提供数据，但无法阻止对方的发送而导致的网络上数据
 - ◆ 即使套接字关闭也不影响对方发出无人接收的数据报

socket控制

■ Socket控制

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

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

`int ioctl(int fd, int cmd, void *arg);`

■ 无阻塞I/O

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

```
int flags;
```

```
flags = fcntl(fd, F_GETFL, 0);
```

```
fcntl(fd, F_SETFL, flags | O_NDELAY);
```

- ◆ 发送缓冲区满，`write`立即以-1返回，`errno`置为EWOULDBLOCK（或EAGAIN）
- ◆ 发送缓冲区半满，`write`返回实际发送的字节数
- ◆ 接收缓冲区空，`read`立即以-1返回，`errno`置为EWOULDBLOCK

单进程并发处理

select: 多路I/O

■ 引入select系统调用的原因

- ◆ 使得用户进程可同时等待多个事件发生
- ◆ 用户进程告知内核多个事件，某一个或多个事件发生时select返回，否则，进程睡眠等待

■ `int select(int maxfdp1, fd_set *rfds, fd_set *wfds, fd_set *efds, struct timeval *timeout);`

- ◆ 例如：告知内核在`rfds`集合{4,5,7}中的任何文件描述符“读准备好”，或在`wfds`集合{3,7}中的任何文件描述符“写准备好”，或在`efds`集合{4,5,8}中的任何文件描述符有“异常情况”发生
- ◆ 集合参数是**传入传出型**，select返回后会被修改，只有准备好的文件描述符，仍出现在集合中
- ◆ 集合参数允许传NULL，表示不关心这方面事件

select: “准备好”

■ 什么叫“准备好”

- ◆ *rfds* 中某文件描述符的 read 不会阻塞
- ◆ *wfds* 中某文件描述符的 write 不会阻塞
- ◆ *efds* 中某文件描述符发生了异常情况
 - TCP 协议，只有加急数据到达才算“异常情况”
 - 对方连接关闭或网络故障，不算“异常情况”

■ “准备好”后可以进行的操作

- ◆ 当“读准备好”时，调用 read 会立刻返回 -1/0/字节数
- ◆ 当“写准备好”时，调用 write 可以写多少字节？
 - ≥ 1 个字节
 - “无阻塞 I/O”方式

集合操作

预定义数据类型fd_set（在C语言头文件定义）

■ **void FD_ZERO(fd_set *fds);**

将*fds*清零：将集合*fds*设置为“空集”

■ **void FD_SET(int fd, fd_set *fds);**

向集合*fds*中加入一个元素*fd*

■ **void FD_CLR(int fd, fd_set *fds);**

从集合*fds*中删除一个元素*fd*

■ **int FD_ISSET(int fd, fd_set *fds);**

判断元素*fd*是否在集合*fds*内

select: 时间

■ select的最后一个参数`timeout`

```
struct timeval {  
    long tv_sec;    /* 秒 */  
    long tv_usec; /* 微秒 */  
};
```

- ◆ 定时值不为0: `select`在某一个描述符I/O就绪时立即返回; 否则等待但不超过`timeout`规定的时限
 - 尽管`timeout`可指定微秒级精度的时间段, 依赖于硬件和软件的设定, 实际实现一般是10毫秒级别
- ◆ 定时值为0: `select`立即返回(无阻塞方式查询)
- ◆ 空指针NULL: `select`等待到至少有一个文件描述符准备好后才返回, 否则无限期地等下去

单进程并发：server2.c(1)

```
int main(void)
{
    int admin_sock, data_sock, ret, n, fd;
    struct sockaddr_in name;
    fd_set fds, rfds;

    admin_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    name.sin_family = AF_INET;
    name.sin_addr.s_addr = INADDR_ANY;
    name.sin_port = htons(PORT_NO);
    bind(admin_sock, (struct sockaddr*)&name, sizeof(name));

    listen(admin_sock, 5);
    printf("ready\n");

    n = admin_sock + 1;
    FD_ZERO(&fds);
    FD_SET(admin_sock, &fds);
```

单进程并发：server2.c(2)

```
for (;;) {
    memcpy(&rfd, &fd, sizeof(fd));
    select(n, &rfd, 0, 0, 0);
    if (FD_ISSET(admin_sock, &rfd)) {
        data_sock = accept(admin_sock, 0, 0);
        FD_SET(data_sock, &fd);
        if (n <= data_sock)
            n = data_sock + 1;
    }
    for (fd = 0; fd < n; fd++) {
        if (fd != admin_sock && FD_ISSET(fd, &rfd)) {
            ret = receive_data(fd);
            if (ret <= 0) {
                close(fd);
                FD_CLR(fd, &fd);
            }
        }
    }
}
```

单进程并发：server2.c(3)

```
static int receive_data(int sock)
{
    unsigned char rbuf[SIZE];
    struct sockaddr_in peer;
    int i, nbyte, name_len = sizeof(peer);
    nbyte = recv(sock, rbuf, SIZE, 0);
    if (nbyte < 0) {
        perror("receiving stream packet");
        return 0;
    }
    getpeername(sock, &peer, &name_len);
    printf("%s:%d ", inet_ntoa(peer.sin_addr), ntohs(peer.sin_port));
    if (nbyte == 0) {
        printf("*** Disconnected.\n");
        return 0;
    }
    for (i = 0; i < nbyte; i++)
        printf("%c", rbuf[i]);
    return 1;
}
```


libevent: 开源跨平台轻量级事件管理库

- ◆ C语言编写的、轻量级的开源高性能事件通知库
- ◆ 事件驱动(event-driven), 高性能
- ◆ 跨平台(Windows/Linux/BSD/MacOS)
- ◆ 支持多种I/O多路复用技术: select, epoll, kqueue等
- ◆ 支持I/O, 定时器和信号等事件
- ◆ 可作为一些应用的底层的网络库

UDP通信

客户端udpclient.c

```
int main(int argc, char *argv[])
{
    int sock, len;
    struct sockaddr_in name;
    unsigned char buf[SIZE];
    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    name.sin_family = AF_INET;
    inet_aton(argv[1], &name.sin_addr);
    name.sin_port = htons(PORT_NO);
    connect(sock, (struct sockaddr *)&name, sizeof(name));
    for(;;) {
        if (fgets(buf, sizeof buf, stdin) == NULL)
            break;
        len = write(sock, buf, strlen(buf));
        if (len < 0) {
            perror("sending message");
            break;
        }
        printf("send %d bytes\n", len);
    }
    close(sock);
}
```

客户端程序udpclient.c(2)

■ connect

- ◆ 不产生网络流量，内核记下远端端点名
- ◆ 之前未用bind指定本地端点名，系统自动分配本地端点名

■ write

- ◆ 使用前面的connect调用指定的端点名
- ◆ UDP不是面向连接的协议，可在sendto参数中指定对方端点名，而且允许对方端点名不同
- ◆ 每次都使用sendto发送数据，前面的connect调用没必要了
- ◆ connect/第一次sendto可使得socket获得系统动态分配的本地端点名，未获得本地端点名之前不该执行read或recv以及recvfrom

服务端程序udpserver.c

```
int main(void)
{
    int sock, len;
    struct sockaddr_in name;
    unsigned char buf[SIZE];

    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    name.sin_family = AF_INET;
    name.sin_addr.s_addr = INADDR_ANY;
    name.sin_port = htons(PORT_NO);
    bind(sock, (struct sockaddr*)&name, sizeof(name));

    for (;;) {
        len = read(sock, buf, sizeof buf);
        if (len < 0) {
            perror("read");
            exit(0);
        }
        printf("Receive %d bytes\n", len);
    }
}
```

UDP通信程序

■ 接收

- ◆ 没有数据到达时，read调用会使得进程睡眠等待
- ◆ 一般需区分数据来自何处，常用recvfrom获得对方的端点名

■ 发送

- ◆ 服务器端发送数据常用sendto，指定远端端点名
- ◆ 对接收来的数据作应答，sendto引用的对方端点名利用recvfrom返回得到的端点名

■ select定时

- ◆ select可实现同时等待两个事件：收到数据和定时器超时
- ◆ 用time(0)或者gettimeofday()获得时间坐标，计算时间间隔决定是否执行超时后的动作

■ 死锁问题

服务端程序udpserver2.c

```
struct sockaddr_in peer;
socklen_t addr_len;
unsigned char buf[SIZE];

for (;;) {
    addr_len = sizeof(peer);
    len = recvfrom(sock, buf, sizeof buf, 0,
                   (struct sockaddr *)&peer, &addr_len);
    if (len < 0) {
        perror("recvfrom");
        continue;
    }
    printf("Received %d bytes from %s:%d\n", len,
           inet_ntoa(peer.sin_addr), ntohs(peer.sin_port));
    sendto(sock, buf, len, 0, (struct sockaddr *)&peer, sizeof peer);
}
```

Socket: 小节

- 协议栈实现：用户态还是核心态
- 三类特殊文件：管道、终端、socket
- Socket机制不仅用于TCP/IP
- TCP/UDP服务模型
- 网络字节顺序问题
- TCP客户服务器
- 端点名的概念
- 客户端程序的几个调用以及进程状态
- 服务器端程序
- 服务器程序的几个调用
- 多进程并发的文件描述符问题
- inetd的实现方法
- TCP服务端设多个socket原因
- read/write与进程状态，以及与TCP协议操作的时序关系
- Socket感知网络故障的手段与TCP协议的对应关系
- 获取端点名系统调用
- read/write的其他版本
- shutdown：功能以及与TCP协议的关系
- 无阻塞I/O
- 单进程并发处理方法
- 准确理解select调用：什么叫“准备好”
- 集合操作
- select的时间参数
- UDP通信