第五章

网络高级编程

Network Advance Programming

讲师：任继梅

课程目标

- ✓了解计算机网络工作的基本原理
  - 五层协议模型
- ✓掌握网络编程常用API
  - socket、dns、address、endian
- ✓TCP编程
  - 服务器程序编写、客户端程序编写
- ✓UDP编程
  - 服务器程序编写、客户端程序编写

课程安排

- ✓ 第一天
  - 上午：计算机网络概述　　下午：TCP/IP协议栈
- ✓ 第二天
  - 上午：Socket编程基础　　下午：TCP&UDP编程基础
- ✓ 第三天
  - 上午：高级网络编程　　下午：高级网络编程续
- ✓ 第四天
  - 上午：网络编程实战　　下午：网络编程实战续
- ✓ 第五天
  - 上午：网络编程实战续　　下午：网络编程实战续

本章目标

- ✓ 结构传输 ★★★
- ✓ 高级函数 ★★★
- ✓ 服务器模型 ★★★
- ✓ 心跳机制 ★★★
- ✓ 广播和组播 ★★★
- ✓ UINX域套接字 ★★★

课前提问

1. TCP客户端流程概述？
2. TCP服务器流程概述？
3. UDP客户端流程概述？
4. UDP服务器流程概述？

知识点1-结构传输

网络中的数据

- 二进制
- 上下文：协议

- 注意协议中采用何种形式表示数据已结束或本次传输数据的大小
- 数据传输形式：
  - 纯文本：经验上多采用”\r\n”表示行结束，”\r\n\r\n”表示全部数据结束；读取时一般一个字节一个字节地读；流量占用较大。不用考虑字节序。
  - 结构型数据（二进制数据）：批量读取；流量占用较小；注意考虑字节序；注意结构体字节对齐。
    - ✓ 定长结构：按固定长度读取。
    - ✓ 不定长结构：结构体里增加表示数据长度的成员。

结构传输

- 定长结构传输

◆示例代码：chapter5/fix_struct/

结构传输

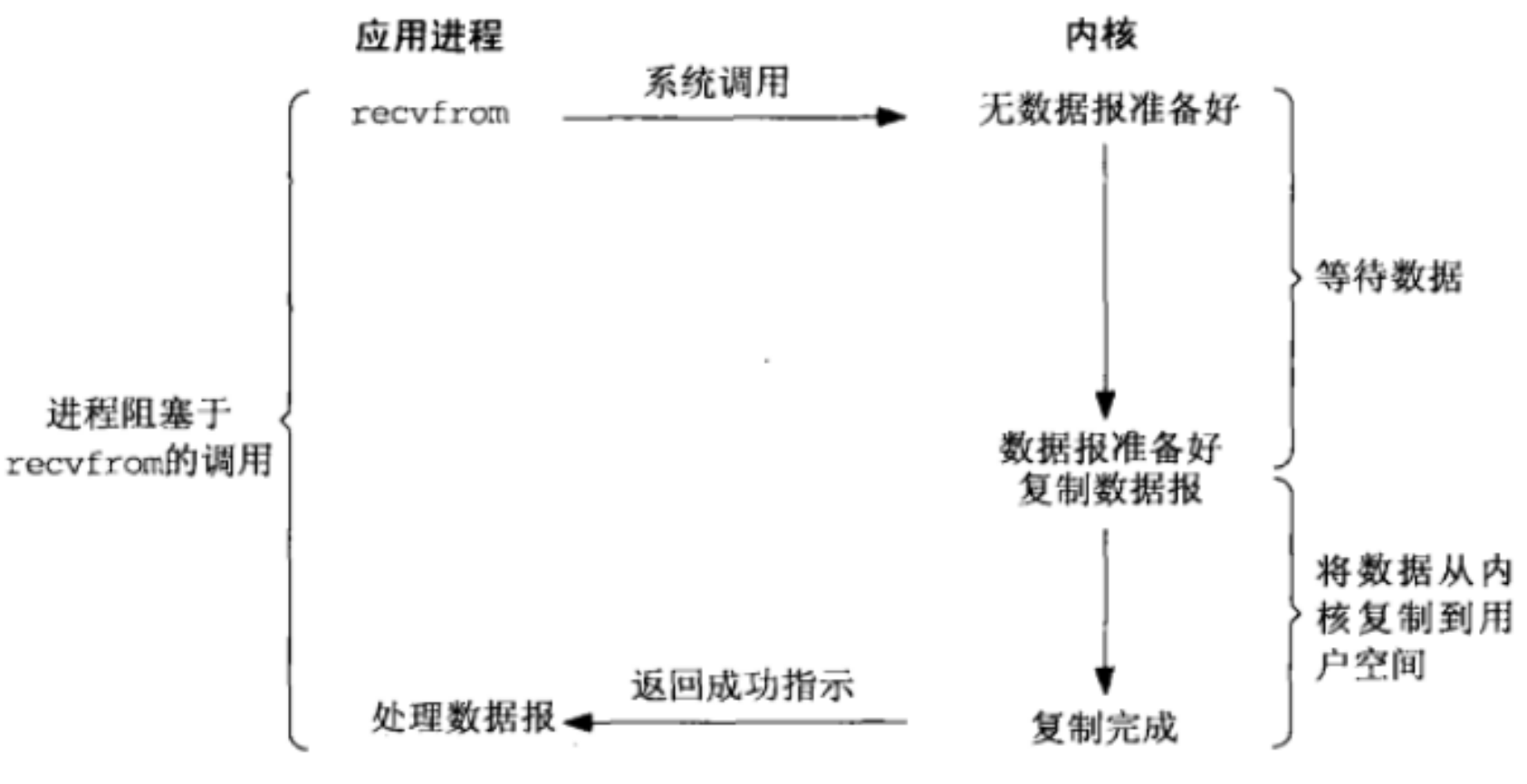
- 不定长结构传输

◆示例代码：chapter5/unfix_struct/

知识点2-高级函数

5种I/O模型——阻塞式I/O

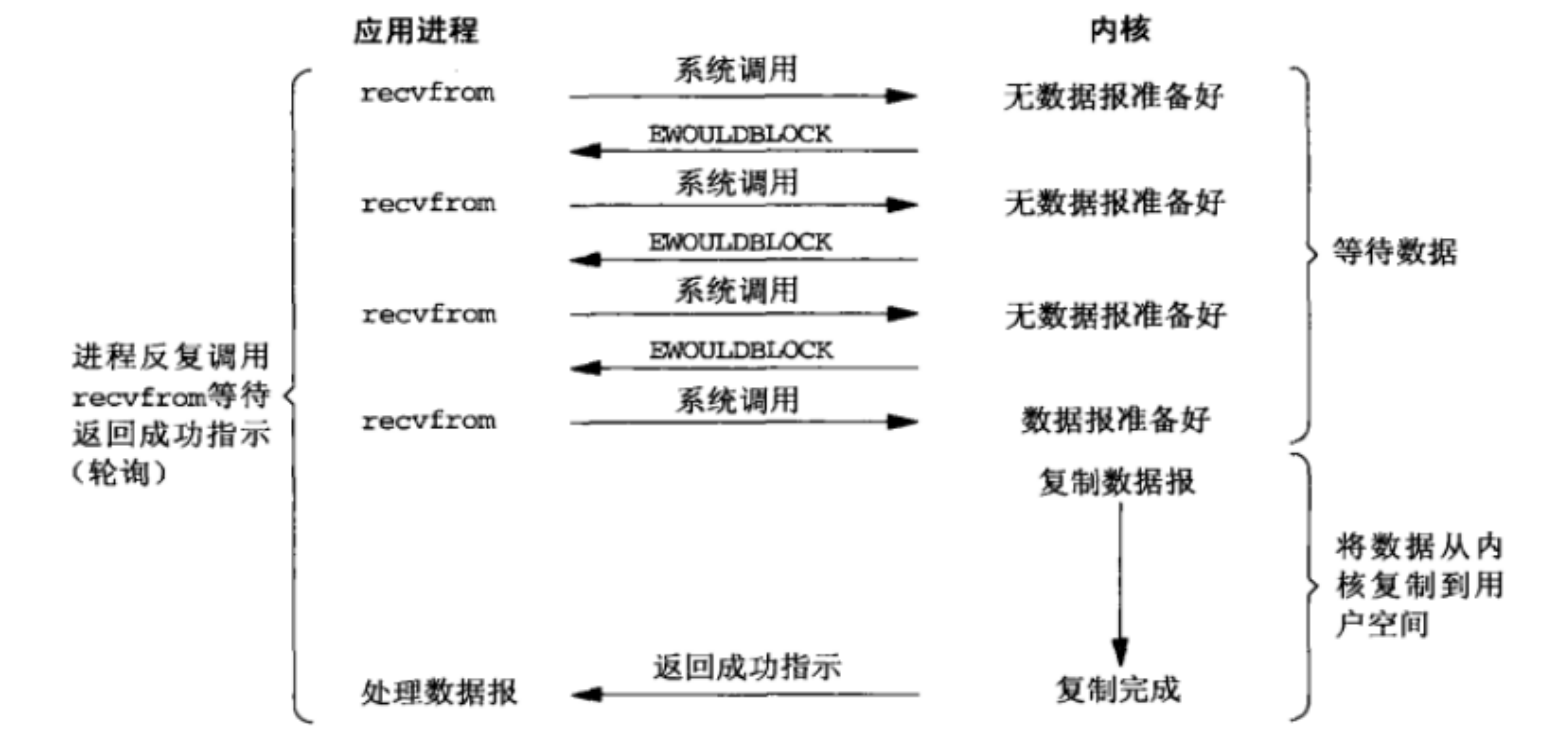
最常用、最简单、效率最低



高级函数

5种I/O模型——非阻塞式I/O

可防止进程阻塞在I/O操作上，需要轮询



高级函数

非阻塞式I/O实现函数

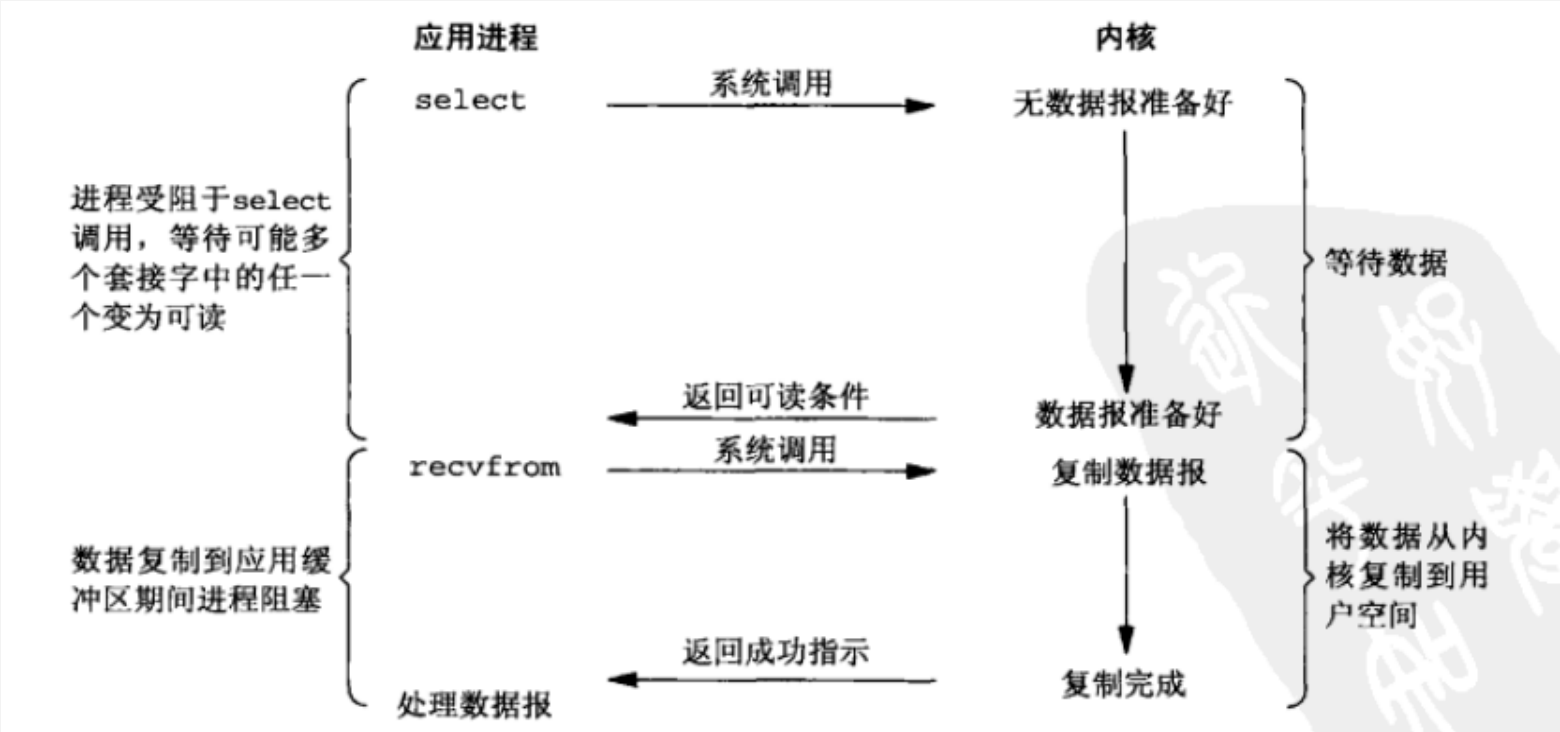
- fcntl()函数
  - 当你一开始建立一个套接字描述符的时候，系统内核将其设置为阻塞IO模式。
  - 可以使用函数fcntl()设置一个套接字的标志为O_NONBLOCK 来实现非阻塞。
  - int fcntl(int fd, int cmd, long arg);

```
int flag;
flag = fcntl(sockfd, F_GETFL, 0);
flag |= O_NONBLOCK;
fcntl(sockfd, F_SETFL, flag);
```

高级函数

5种I/O模型——I/O复用

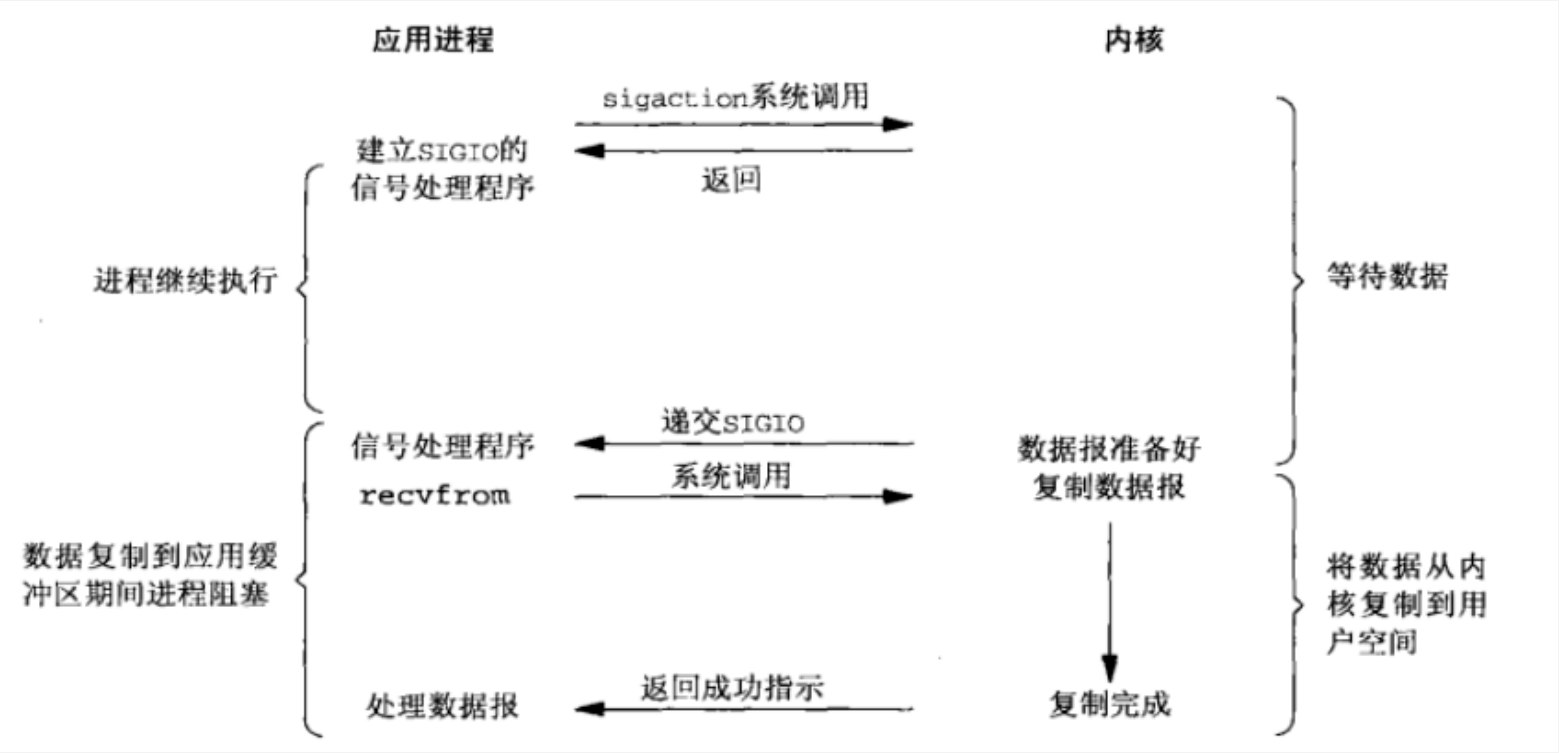
允许同时对多个I/O进行控制



高级函数

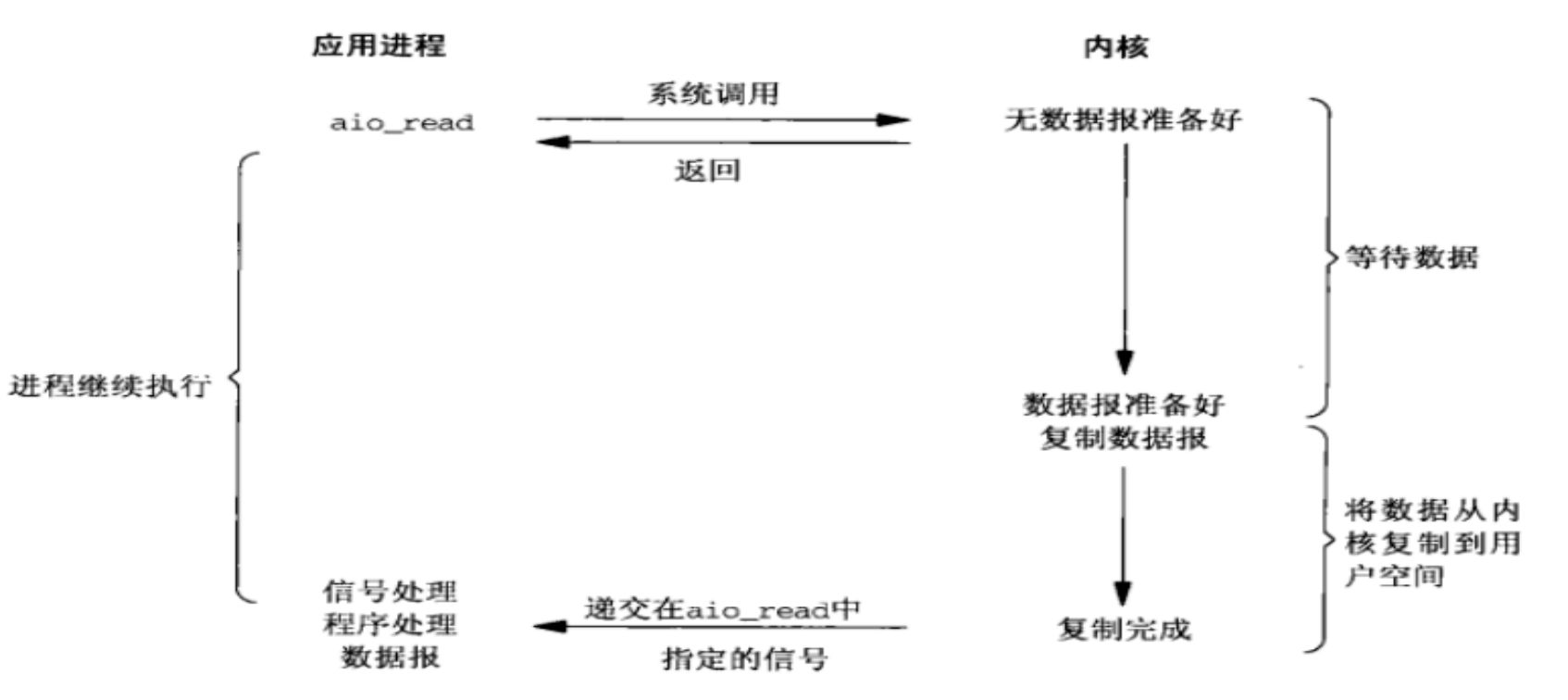
5种I/O模型——信号驱动式I/O

一种异步通信模型



高级函数

- 5种I/O模型——异步I/O



高级函数

多路复用

- 应用程序中同时处理多路输入输出流，若采用阻塞模式，将得不到预期的目的；
- 若采用非阻塞模式，对多个输入进行轮询，但又太浪费CPU时间；
- 若设置多个进程，分别处理一条数据通路，将新产生进程间的同步与通信问题，使程序变得更加复杂；
- 比较好的方法是使用I/O多路复用。其基本思想是：
  - 先构造一张有关描述符的表，然后调用一个函数。当这些文件描述符中的一个或多个已准备好进行I/O时函数才返回。
  - 函数返回时告诉进程那个描述符已就绪，可以进行I/O操作。

高级函数

多路复用

```
#include <sys/select.h>
#include <sys/time.h>

int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
           const struct timeval *timeout);
```

返回：若有就绪描述符则为其数目，若超时则为0，若出错则为-1

该函数允许进程指示内核等待多个事件中的任何一个发生，并只在有一个或多个事件发生或经历一段指定的时间后才唤醒它。

也就是说，我们调用select告知内核对哪些描述符（就读、写或异常条件）感兴趣以及等待多长时间。我们感兴趣的描述符不局限于套接字，任何描述符都可以使用select来测试。

高级函数

select续

```
struct timeval {
  long    tv_sec;     /* seconds */
  long    tv_usec;    /* microseconds */
};
```

这个参数有以下三种可能。

(1) 永远等待下去：仅在有一个描述符准备好I/O时才返回。为此，我们把该参数设置为空指针。

(2) 等待一段固定时间：在有一个描述符准备好I/O时返回，但是不超过由该参数所指向的timeval结构中指定的秒数和微秒数。

(3) 根本不等待：检查描述符后立即返回，这称为轮询（polling）。为此，该参数必须指向一个timeval结构，而且其中的定时器值（由该结构指定的秒数和微秒数）必须为0。

*maxfdp1*参数指定待测试的描述符个数，它的值是待测试的最大描述符加1（因此我们把该参数命名为*maxfdp1*），描述符0, 1, 2…一直到*maxfdp1*–1均将被测试。

高级函数

- select()/poll()实现多路复用

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

int select(int n, fd_set *read_fds, fd_set *write_fds, fd_set
*except_fds, struct timeval *timeout);

#include <sys/poll.h>
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

高级函数

Select参数

- maxfd
  - 所有监控的文件描述符中最大的那一个加1
- read_fds
  - 所有要读的文件文件描述符的集合
- write_fds
  - 所有要的写文件文件描述符的集合
- except_fds
  - 其他要向我们通知的文件描述符
- timeout
  - 超时设置.
  - Null：一直阻塞，直到有文件描述符就绪或出错
  - 时间值为0：仅仅检测文件描述符集的状态，然后立即返回
  - 时间值不为0：在指定时间内，如果没有事件发生，则超时返回。

高级函数

fd_set配套操作

- 在我们调用select时进程会一直阻塞直到以下的一种情况发生.

```
fd_set  rset;
```

  - 有文件可以读.
  - 有文件可以写.
  - 超时所设置的时间到.
- 为了设置文件描述符我们要使用几个宏：
  - **FD_SET**　将fd加入到fdset
  - **FD_CLR**　将fd从fdset里面清除
  - **FD_ZERO**　从fdset中清除所有的文件描述符
  - **FD_ISSET**　判断fd是否在fdset集合中

课堂案例

Select的使用

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define  N  64

typedef struct sockaddr  SA;

int  main(int argc,  char *argv[])
{
   int i, listenfd, connfd, maxfd;
   char buf[N];
   fd_set  rdfs;
```

Select使用

```
struct sockaddr_in myaddr;

 if  ((listenfd = socket(PF_INET,
   SOCK_STREAM, 0)) < 0)
 {
    perror("fail to socket");
    exit(-1);
 }

  bzero(&myaddr, sizeof(myaddr));
  myaddr.sin_family = PF_INET;
  myaddr.sin_port = htons(8888);
  myaddr.sin_addr.s_addr =
    inet_addr("127.0.0.1");

  if (bind(listenfd, (SA *)&myaddr,
    sizeof(myaddr)) < 0)
  {
    perror("fail to bind");
    exit(-1);
  }
```

课堂案例

Select的使用

```
listen(listenfd, 5);
    maxfd = listenfd;

    while ( 1 )
    {
        FD_ZERO(&rdfs);
        FD_SET(0, &rdfs);
        FD_SET(listenfd, &rdfs);

        if (select(maxfd+1, &rdfs, NULL,
     NULL, NULL) < 0)
        {
            perror("fail to select");
            exit(-1);
        }

         for(i=0; i<=maxfd; i++)
         {
             if (FD_ISSET(i, &rdfs))
             {
```

Select使用

```
 if (i = = STDIN_FILENO)
                {
                    fgets(buf, N, stdin);
                    printf("%s", buf);
                }
                else if (i = = listenfd)
                {
                    connfd = accept(listenfd,
     NULL, NULL);
                    printf("New Connection
     connfd is coming\n", connfd);
                    close(connfd);
                }
            }
        } //  end for
    } // end while

    return 0;
}
```

高级函数

getsockname/getpeername

这两个函数或者返回与某个套接字关联的本地协议地址（getsockname），或者返回与某个套接字关联的外地协议地址（getpeername）。

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);
```

均返回：若成功则为0，若出错则为-1

注意，这两个函数的最后一个参数都是值-结果参数。这就是说，这两个函数都得装填由*localaddr*或*peeraddr*指针所指的套接字地址结构。

高级函数

- 套接字选项——了解

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

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

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

均返回：若成功则为0，若出错则为-1

知识点3-服务器模型

服务器常见模型

在网络程序里面，通常都是一个服务器处理多个客户机。

为了处理多个客户机的请求， 服务器端的程序有不同的处理方式。

目前最常用的服务器模型.

循环服务器：

循环服务器在同一个时刻只能响应一个客户端的请求

并发服务器：

并发服务器在同一个时刻可以响应多个客户端的请求

服务器模型

循环服务器模型

- **TCP服务器**
  - TCP服务器端运行后等待客户端的连接请求。
  - TCP服务器接受一个客户端的连接后开始处理，完成了客户的所有请求后断开连接。
  - TCP循环服务器一次只能处理一个客户端的请求。
  - 只有在当前客户的所有请求都完成后，服务器才能处理下一个客户的连接/服务请求。
  - 如果某个客户端一直占用服务器资源，那么其它的客户端都不能被处理。TCP服务器一般很少采用循环服务器模型。

服务器模型

- TCP循环服务器

流程如下:

```
socket(...);
bind(...);
listen(...);
while(1)
{
    accept(...);
    while(1)
    {
        recv(...);
        process(...);
        send(...);
    }
    close(...);
}
```

服务器模型

UDP循环服务器

- **UDP服务器**
  - UDP服务器每次从套接字上读取一个客户端的请求，处理后将结果返回给客户机.
  - 可以用下面的算法来实现(伪代码):

```
socket(...);
bind(...);
while(1)
{
    recvfrom(...);
    process(...);
    sendto(...);
}
```

服务器模型

TCP并发服务器模型

- **TCP服务器**
  - 为了弥补TCP循环服务器的缺陷，人们又设计了并发服务器的模型。

    并发服务器的设计思想是服务器接受客户端的连接请求后创建子进程来为客户端服务
  - TCP并发服务器可以避免TCP循环服务器中客户端独占服务器的情况。
  - 为了响应客户机的请求,服务器要创建子进程来处理。 如果有多个客户端的话，服务器端需要创建多个子进程。过多的子进程会影响服务器端的运行效率

服务器模型

- TCP并发服务器
  - 流程如下:

```
socket(...);
bind(...);
listen(...);
while(1) {
    accept(...);
    if (fork() = = 0) {
        while(1) { recv(...); process(...); send(...); }
        close(...);
        exit(...);
    }
    close(...);
}
```

服务器模型

UDP并发服务器

- **UDP服务器**
  - 人们把并发的概念用于UDP就得到了UDP并发服务器模型。
  - UDP并发服务器模型TCP服务器模型一样，创建一个子进程来处理客户端的请求
  - 除非UDP服务器在处理某个客户端的请求时所用的时间比较长,人们实际上较少用这种模型。

服务器模型

I/O多路复用并发服务器

- **I/O多路复用并发服务器**

```
初始化(socket -> bind -> listen);
 while(1) {
    设置监听读写文件描述符集合(FD_*);
    调用select;
    如果是监听套接字就绪,说明有新的连接请求 {
       建立连接(accept);
       加入到监听文件描述符集合;
     }
    否则说明是一个已经连接过的描述符 {
       进行操作(send或者recv);
    }
 }
```

服务器模型

I/O多路复用并发服务器

- I/O多路复用模型可以解决资源限制的问题。此模型实际上是将UDP循环模型用在了TCP上面。
- 服务器使用单进程循环处理请求（客户端有限的情况下）。
- 存在同样的问题：由于服务器是依次处理客户的请求，所以可能会导致有的客户等待时间过长。

知识点4-心跳机制

网络监测的必要性

- 在网络通信中，很多操作会使得进程阻塞
- TCP套接字中的recv/accept/connect
- UDP套接字中的recvfrom
- 超时检测的必要性
  - 避免进程在没有数据时无限制地阻塞
  - 当设定的时间到时，进程从原操作返回继续运行

心跳机制

网络超时监测

- 设置socket的属性 SO_RCVTIMEO
- 参考代码如下

```
struct timeval  tv;

tv.tv_sec = 5;  //  设置5秒时间
tv.tv_usec = 0;

setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO,  &tv,
                sizeof(tv));  //  设置接收超时
 recv() / recvfrom()   //   从socket读取数据
```

心跳机制

网络超时检测

- 用select检测socket是否’ ready’
- 参考代码如下

```
struct fd_set rdfs;
struct timeval  tv = {5 , 0};  // 设置5秒时间

FD_ZERO(&rdfs);
FD_SET(sockfd, &rdfs);

if (select(sockfd+1, &rdfs, NULL, NULL, &tv) > 0)  // socket就绪
{
    recv() /  recvfrom()  //  从socket读取数据
}
```

心跳机制

网络超时检测

- 设置定时器(timer), 捕捉SIGALRM信号
- 参考代码如下

```
void  handler(int signo)    {   return;  }

 struct sigaction  act;
 sigaction(SIGALRM, NULL, &act);
 act.sa_handler = handler;
 act.sa_flags &= ~SA_RESTART;
 sigaction(SIGALRM, &act, NULL);
 alarm(5);
 if (recv(,,,) < 0) ……
```

知识点5-广播和组播

广播

- 数据包发送方式只有一个接受方，称为单播
- 如果同时发给局域网中的所有主机，称为广播
- 只有用户数据报(使用UDP协议)套接字才能广播
- 广播地址
  - 以192.168.1.0（255.255.255.0）网段为例，最大的主机地址192.168.1.255代表该网段的广播地址
  - 发到该地址的数据包被所有的主机接收
  - 255.255.255.255在所有网段中都代表广播地址

广播和组播

广播发送

- 创建用户数据报套接字
- 缺省创建的套接字不允许广播数据包，需要设置属性
  - setsockopt可以设置套接字属性
- 接收方地址指定为广播地址
- 指定端口信息
- 发送数据包

广播和组播

setsocopt

- int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
  - 头文件：<sys/socket.h>
  - level : 选项级别（例如SOL_SOCKET）
  - optname : 选项名（例如SO_BROADCAST）
  - optval : 存放选项值的缓冲区的地址
  - optlen : 缓冲区长度
  - 返回值：成功返回0 失败返回-1并设置errno

广播和组播

- 广播发送实例

```
sockfd = socket(,,);
......
int on = 1;
setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on,
    sizeof(on));
......
sendto(;;;;;);
```

广播和组播

广播接收

- 创建用户数据报套接字
- 绑定本机IP地址和端口
  - 绑定的端口必须和发送方指定的端口相同
- 等待接收数据

广播和组播

组播

- 单播方式只能发给一个接收方。
- 广播方式发给所有的主机。过多的广播会大量占用网络带宽，造成广播风暴，影响正常的通信。
- 组播(又称为多播)是一种折中的方式。只有加入某个多播组的主机才能收到数据。
- 多播方式既可以发给多个主机，又能避免象广播那样带来过多的负载(每台主机要到传输层才能判断广播包是否要处理)

广播和组播

网络地址

- A类地址
  - 第1字节为网络地址，其他3个字节为主机地址。第1字节的最高位固定为0
  - 1.0.0.1 – 126.255.255.255
- B类地址
  - 第1字节和第2字节是网络地址，其他2个字节是主机地址。第1字节的前两位固定为10
  - 128.0.0.1 – 191.255.255.255
- C类地址
  - 前3个字节是网络地址，最后1个字节是主机地址。第1字节的前3位固定为110
  - 192.0.0.1 – 223.255.255.255
- D类地址（组播地址）
  - 不分网络地址和主机地址，第1字节的前4位固定为1110
  - 224.0.0.1 – 239.255.255.255

广播和组播

组播发送

- 创建用户数据报套接字
- 接收方地址指定为组播地址
- 指定端口信息
- 发送数据包

广播和组播

组播接收

- 创建用户数据报套接字
- 加入多播组
- 绑定本机IP地址和端口
  - 绑定的端口必须和发送方指定的端口相同
- 等待接收数据

广播和组播

加入多组播

```
struct ip_mreq
{
    struct  in_addr  imr_multiaddr;
    struct  in_addr  imr_interface;
};

struct  ip_mreq  mreq;
bzero(&mreq, sizeof(mreq));
mreq.imr_multiaddr.s_addr = inet_addr("224.10.10.1");
mreq.imr_interface.s_addr = htonl(INADDR_ANY);

setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq,
            sizeof(mreq));
```

知识点6-Unix域套接字

本地Socket

- socket同样可以用于本地通信
- 创建套接字时使用本地协议PF_UNIX(或PF_LOCAL)。
- 分为流式套接字和用户数据报套接字
- 和其他进程间通信方式相比使用方便、效率更高
- 常用于前后台进程通信

Unix域套接字

本地地址结构

本地地址结构

```
struct sockaddr_un       //  <sys/un.h>
{
    sa_family_t  sun_family;
    char  sun_path[108];       // 套接字文件的路径
};
```

填充地址结构

```
struct sockaddr_un myaddr;
bzero(&myaddr,  sizeof(myaddr));
myaddr.sun_family = PF_UNIX;
strcpy(myaddr.sun_path,  “mysocket”);
```

Unix域套接字

流式套接字

- 服务器端

socket(PF_UNIX, SOCK_STREAM, 0)

bind(，本地地址，)

listen(，)

accept(，，)

recv() / send()

.....

Unix域套接字

流式套接字

- 客户端

```
socket(PF_UNIX, SOCK_STREAM, 0)

bind(，本地地址，）// 可选

connect(，，)

recv() / send()

.....
```

Unix域套接字

用户数据报套接字

- 服务器端

socket(PF_UNIX, SOCK_DGRAM, 0)

bind(，本地地址，)

recvfrom()

sendto()

Unix域套接字

用户数据报套接字

- 客户端

socket(PF_UNIX, SOCK_DGRAM, 0)

bind(，本地地址，) // 可选

sendto()

recvfrom() // 若没有绑定地址，无法接收数据

……

Unix域套接字

常用调试工具

- 使用telnet测试TCP服务器端
- 使用lsof
- 使用netstat
- 使用sniffer(tcpdump, ethereal, etc.)
- wireshark

联系方式

上款

课程总结

- 本节课程内容
  - 结构传输
  - 高级函数
  - 服务器模式
  - 广播和组播
  - 心跳机制
  - 本地Socket
- 下节课程
  - 网络编程实践