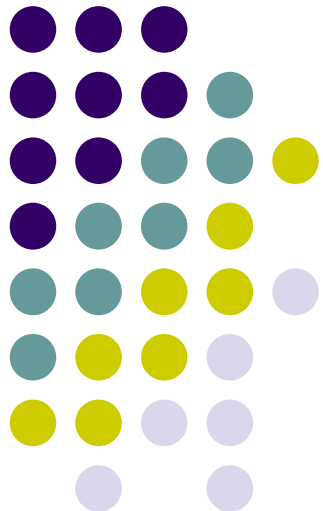
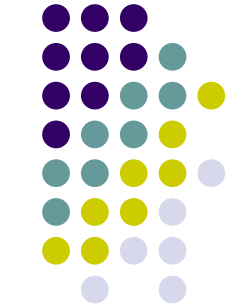
Ch6 并发服务器模型



主要内容



- 并发服务器设计

Part 1 并发服务器设计

- 1.1 迭代服务器
- 1.2 简单并发服务器 ——单客户单进程
- 1.3 单线程服务器——单客户单线程
- 1.4 I/O复用服务器
- 1.5 进程池
- 1.6 线程池

1.1 迭代服务器

- 接受一个客户端的连接，然后处理，完成了这个客户的所有请求后，断开连接。TCP循环服务器一次只能处理一个客户端的请求，只有在这个客户的所有请求满足后，服务器才可以继续后面的请求。如果有一个客户端占住服务器不放时，其它的客户机都不能工作了。

1.1 迭代服务器——代码结构

程序代码结构：

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

1.1 迭代服务器——特点

在accept之后，就开始在这一个连接连接上的数据接收，收到之后处理，发送，不再接收新的连接，除非这个连接的处理结束。

优点：

简单。

缺点：

因为只为一个客户端服务，所以不存在并发的可能。

应用：

用在只为一个客户端服务的时候。

1.1 迭代服务器——举例

例5-8-0client.c

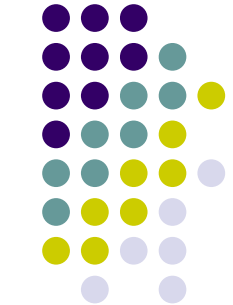
```
sockfd = socket(AF_INET, SOCK_STREAM, 0)
 ……
connect(sockfd,(struct
 sockaddr*)&s_addr,sizeof(struct sockaddr)
 ……
write(sockfd,buf,strlen(buf));
 ……
len = read(sockfd,buf,BUFLEN);buf[len]=0;
  if(len > 0)       printf("服务器的系统时间是：%d
 %s\n",len,buf);
close(sockfd);
 ……
```

1.1 迭代服务器——举例

例5-8-1server.c

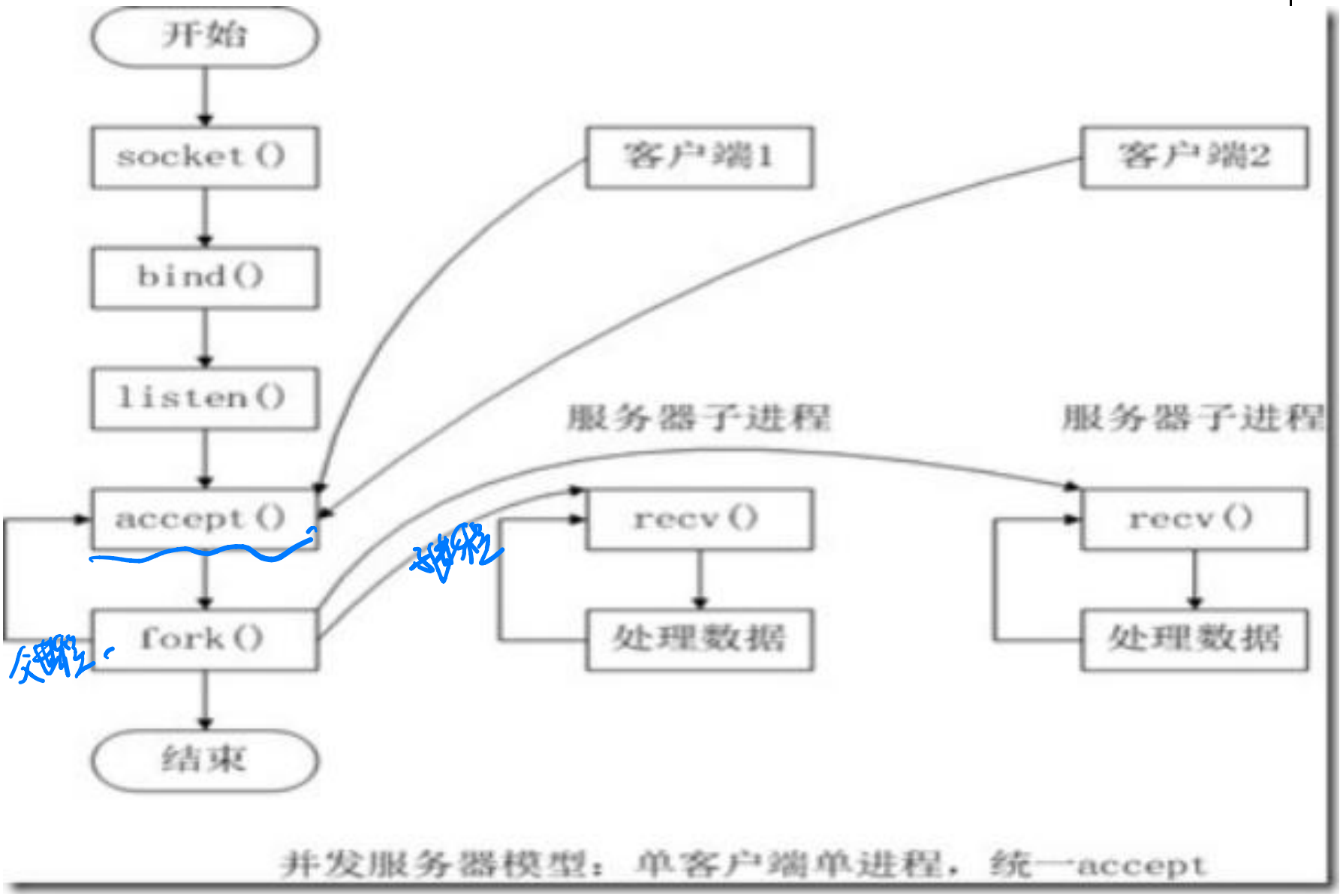
```
int main(void)
{   sock_descriptor = socket(AF_INET, SOCK_STREAM, 0);
        ……
    bind(sock_descriptor, (struct sockaddr*)&sin, sizeof(sin);
    listen(sock_descriptor, 100) ;
    while(1) {      ……
        temp_sock_descriptor = accept(sock_descriptor, (struct
    sockaddr *)&pin, &address_size);
        recv(temp_sock_descriptor, buf, 16384, 0) ;
        ……
        ticks = time(NULL);
        snprintf(buf, sizeof(buf), "%.24s\r\n", ctime(&ticks));
        len=write(temp_sock_descriptor, buf, strlen(buf));
        close(temp_sock_descriptor);
    }
}
```

1.2 简单并发服务器——单客户单进程



当客户有请求时，为每一个客户请求fork一个子进程，一个子进程（线程）处理一个客户端连接请求，父进程监听

1.2 简单并发服务器——模型



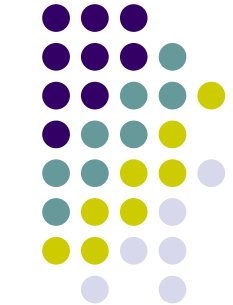
并发服务器模型：单客户端单进程，统一accept

1.2 简单并发服务器——代码结构

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

子进程关闭监听字

1.2 简单并发服务器——特点



- 优点：
- 编程相对简单，不用考虑线程间的数据同步等。
- 缺点：
- 资源消耗大。启动一个进程消耗相对比启动一个线程要消耗大很多，同时在处理很多的连接时候需要启动很多的进程多去处理，这时候对系统来说压力就会比较大。另外系统的进程数限制也需要考虑。
- 应用：
- 在客户端数据不多的时候使用很方便，比如小于10个客户端。

1.2 简单并发服务器——举例

例5-9process.c

```
int main(int argc, char **argv)
{
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  ……
  bind(sockfd, (struct sockaddr*) &s_addr,sizeof(struct
    sockaddr);
  listen(sockfd,listnum) ;
  handle_connect(sockfd);
  close(sockfd);
   return 0;
}
```

1.2 简单并发服务器——举例

```
static void handle_connect(int sockfd){
    ……
    while(1){
       if((newfd = accept(sockfd,(struct sockaddr*) &c_addr, &len)) == -
     1){
            perror("accept");                exit(errno);        }
       else{
                ……
                if(fork() > 0)              close(newfd);
                else{
                    ……
                    handle_request(newfd);
                 }
           }
      }
}
```

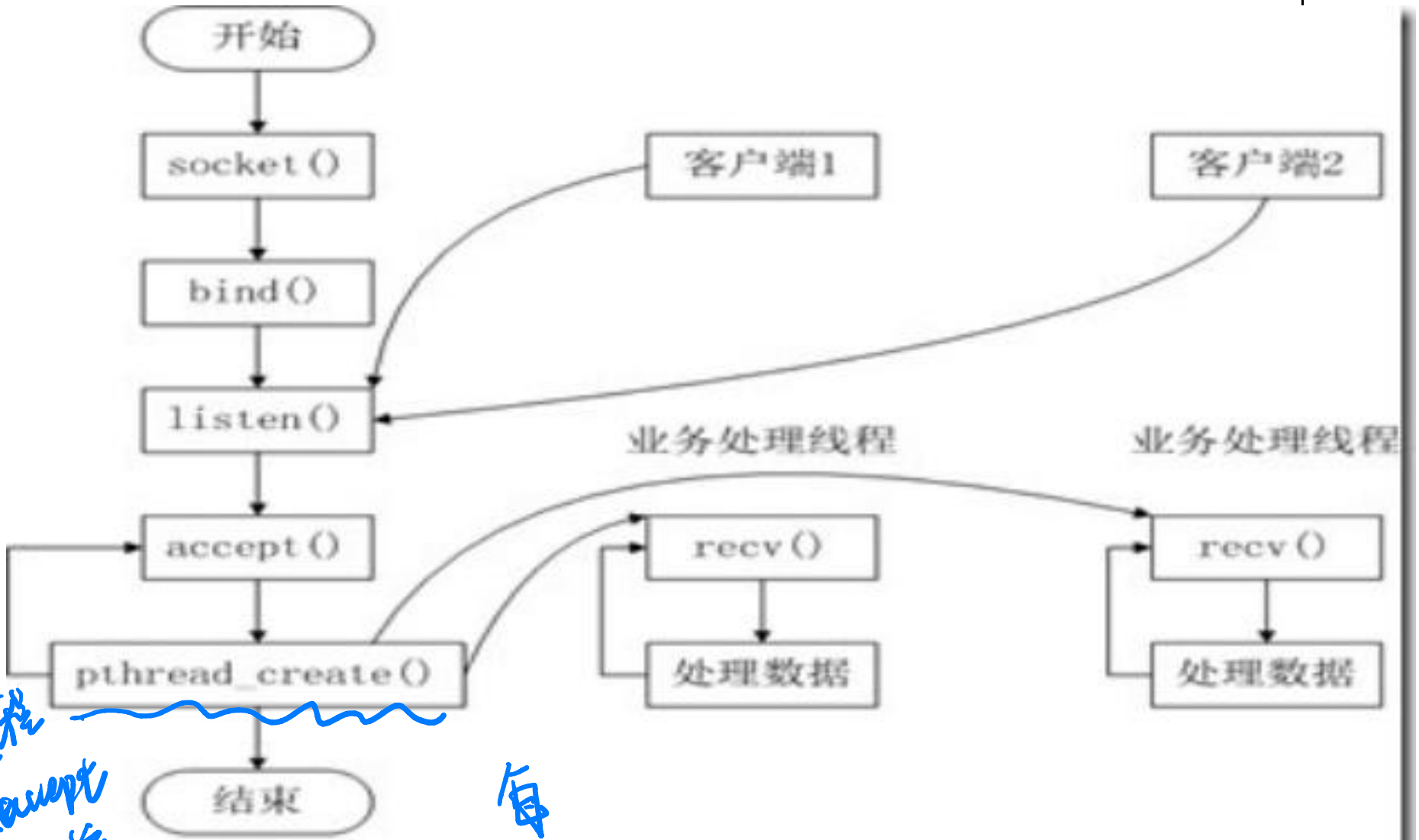
1.2 简单并发服务器——举例

```
static void handle_request(int newfd){
    ……
    len = read(newfd,buf,BUFLEN);
    if(len >0){        ……
        now = time(NULL);
        sprintf(buf,"%24s\r\n",ctime(&now));
        send(newfd,buf,strlen(buf),0);
    }
     close(newfd);
}
```

1.3 单线程服务器

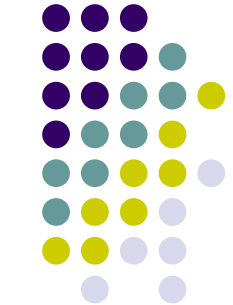
- 在一个主程序中，接收客户端的连接，当客户端连接到来时，使用pthread_create函数建立一个线程进程客户端的请求处理，分析数据，给出响应等。

1.3 单线程服务器——模型



并发服务器模型：单客户端单线程，统一accept

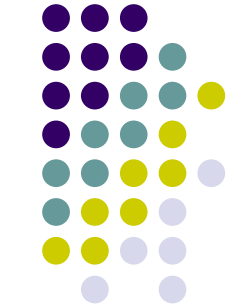
1.3 单线程服务器——代码结构



程序代码结构：

```
socket(...);
bind(...);
listen(...);
while(1)
    {
     accpet(...);
     pthread_create(...);
     ...
      close(...);
    }
```

1.3 单线程服务器——特点



- 为每个客户连接创建一个线程
- 优点：
- 　　相对多进程方式，会节约一些资源，会更加高效一些。
- 缺点：
- 　　相对多进程方式，增加了编程的复杂度，因为需要考虑数据同步和锁保护。另外一个进程中不能启动太多的线程。在Linux系统下线程在系统内部其实就是进程，线程调度按照进程调度的方式去执行的。
- 应用：
- 　　类似于多进程方式，适用于少量的客户端的时候。

1.3 单线程服务器——举例

例5-10thread.c

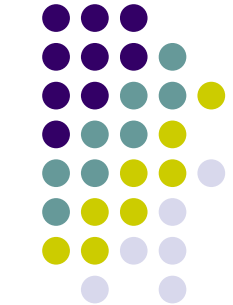
```
int main(int argc, char **argv)
{    ……
     sockfd = socket(AF_INET, SOCK_STREAM, 0)
      ……
    bind(sockfd, (struct sockaddr*) &s_addr,sizeof(struct
     sockaddr)
    listen(sockfd,listnum);
    handle_connect(sockfd);
    close(sockfd);
     return 0;
}
```

1.3 单线程服务器——举例

```
static void handle_connect(int sockfd){
    ……
    pthread_t thread_s;
    while(1){
        ……
        if((newfd = accept(sockfd,(struct sockaddr*)
    &c_addr, &len)) >0){
            ……

    pthread_create(&thread_s,NULL,handle_request,(voi
    d *)&newfd);
        }
    }
}
```

1.3 单线程服务器——举例



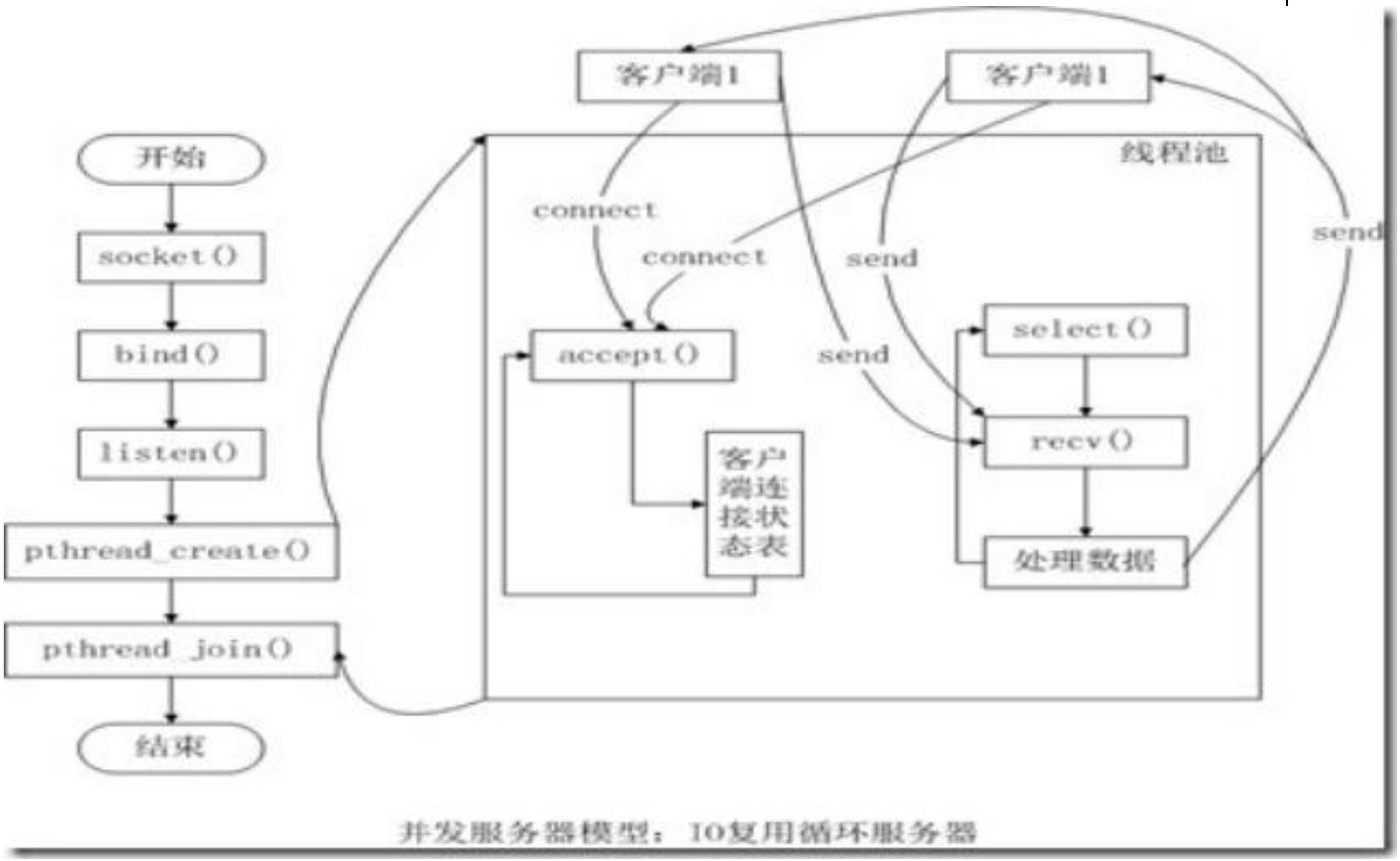
```
static void *handle_request(void *argv){
    ……
    len = read(newfd,buf,BUFLEN);
    if(len >0 ){
        ……
        now = time(NULL);
        sprintf(buf,"%24s\r\n",ctime(&now));
        send(newfd,buf,strlen(buf),0);
    }
    close(newfd);
    return NULL;
}
```

1.4 I/O复用服务器

- I/ O复用技术是为了解决进程或线程阻塞到某个 I/ O系统调用而出现的技术 ,使进程不阻塞于某个特定的I/ O系统调用。它也可用于并发服务器的设计，常用函数select 或 poll来实现。

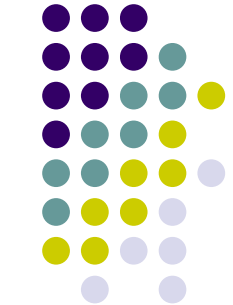
轮询技术

1.4 I/O复用服务器——模型



并发服务器模型：IO复用循环服务器

1.4 I/O复用服务器——代码结构



```
bind(listenfd);
listen(listenfd);
FD_ZERO(&rset);
for(;;){
        clifd = accept();
        cliarray[] = clifd;
   }
  for(;;){
      ……
       FD_SET( cliarray[i],&rset);
       select(...);
       if (FD_ISSET(cliarray[i] , &rset))
       dosomething();
  }
```

Select线程

- 有一个线程专门用于监听端口，accept返回之后就把这个描述符放入 描述符集合 fd中，一个线程用select去轮询描述符集合，在有数据的连接上接收数据，另外一个线程专门发送数据。当然也可以接收和发送用一个线程。描述符可以设置成非阻塞模式，也可以设置成阻塞模式。通常连接设置成非阻塞模式，发送线程独立出来。
- 优点：
- 相对前几种模式，这种模式大大提高了并发量。
- 缺点：
- 系统一般实现描述符集合是采用一个大数组，每次调用select的时候都会轮询这个描述符数组，当连接数很多的时候就会导致效率下降。连接数在1000以上时候效率会下降到不能接受。
- 应用：
- 目前windows 和一般的Unix上的tcp并发都采用select方式，应该说应用还是很广泛的。

1.4 I/O复用服务器——举例

例5-11IO_multiplex.c

```
int main(int argc, char **argv)
{    ……
    sockfd = socket(AF_INET, SOCK_STREAM, 0)
        ……
    bind(sockfd, (struct sockaddr*) &s_addr,sizeof(struct sockaddr));
    listen(sockfd,listnum) ;
    pthread_create(&thread_s[0],NULL,handle_connect,(void *)&sockfd);
    pthread_create(&thread_s[1],NULL,handle_request, NULL);
    for(i = 0; i < THREADNUM; i++){
        pthread_join(thread_s[i],NULL);
      }
   close(sockfd);
    return 0;
}
```

1.4 I/O复用服务器——举例

```
static void *handle_connect(void *arg){
    ……
    while(1){
        ……
        if((newfd = accept(sockfd,(struct sockaddr*) &c_addr, &len)) >0){
            ……
            for(i = 0; i < CLIENTNUM; i++){
                if( connect_host[i] == -1){
                    connect_host[i] = newfd;
                    /*客户端计数器*/
                    connect_num++;
                    /*继续等待新的客户端*/
                    break;
                }       }       }   }
    return NULL;
}
```

1.4 I/O复用服务器——举例

```
static void *handle_request(void *argv){
    …
    int maxfd = -1;
    fd_set rfds;
    struct timeval tv;
    tv.tv_sec = 1;
    tv.tv_usec = 0;
    …
    while(1){
        FD_ZERO(&rfds);
        for(i = 0; i < CLIENTNUM; i++){
            if(connect_host[i] != -1){
                FD_SET(connect_host[i],&rfds);
                if(maxfd < connect_host[i])
                    maxfd = connect_host[i];
            }
        }
```

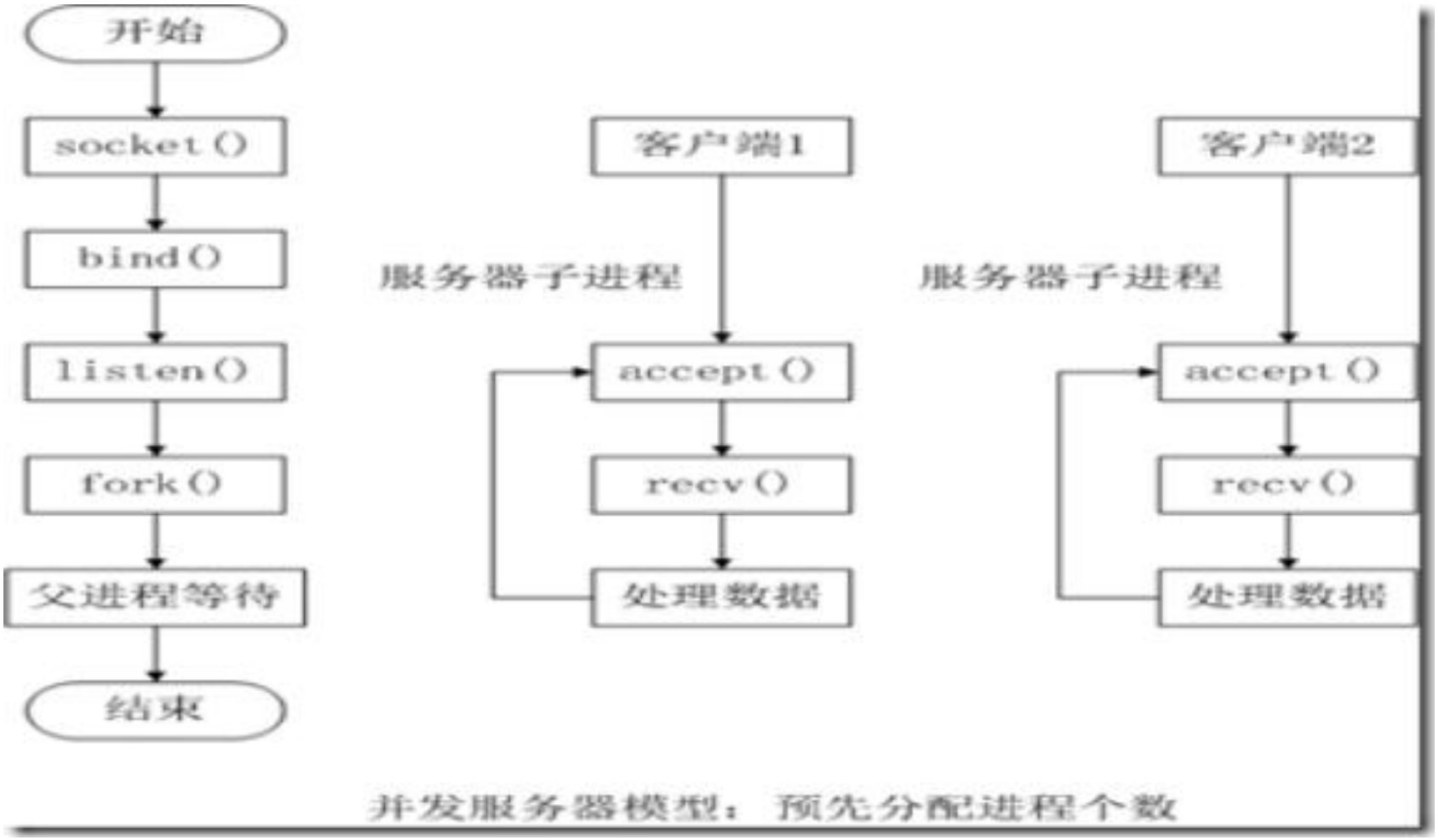
1.4 I/O复用服务器——举例

```
err = select(maxfd+1, &rfds, NULL, NULL, &tv);
switch(err){
    case 0: break;
    case -1: break;
    default:
      if (connect_num < 0)                    break;
      for(i = 0; i < CLIENTNUM; i++){
            if(connect_host[i] != -1){
                  if(FD_ISSET(connect_host[i],&rfds)){
                  len = read(connect_host[i],buf,BUFLEN);
                  if(len >0){                              …
                     send(connect_host[i],buf,strlen(buf),0);
                     }
                  close(connect_host[i]);
                  connect_host[i] = -1; connect_num--;
                }                 }                 }                 break;
    }    }    return NULL;
}
```

1.5 进程池

- 进程池
- 方法：在服务器端，主程序提前构建多个子进程，当客户端的请求到来的时候，系统从进程池中选取一个子进程来处理客户端的连接，每个子进程处理一个客户端的请求。具体模型为如下：

1.5 进程池——模型


开始
socket()
bind()
listen()
fork()
父进程等待
结束
客户端1
服务器子进程
accept()
recv()
处理数据
客户端2
服务器子进程
accept()
recv()
处理数据
并发服务器模型：预先分配进程个数

1.5 进程池——举例

例5-12process_poll.c

```
int main(int argc, char **argv)
{    ……
    sockfd = socket(AF_INET, SOCK_STREAM, 0)
     ……
    bind(sockfd, (struct sockaddr*) &s_addr,sizeof(struct sockaddr));
    ……
    listen(sockfd,listnum)
    int i = 0;
    for(i = 0; i < PIDNUM; i++){
        pid[i] = fork();
        if(pid[i] == 0)
            handle_fork(sockfd);       }
    close(sockfd);
    return 0;
}
```

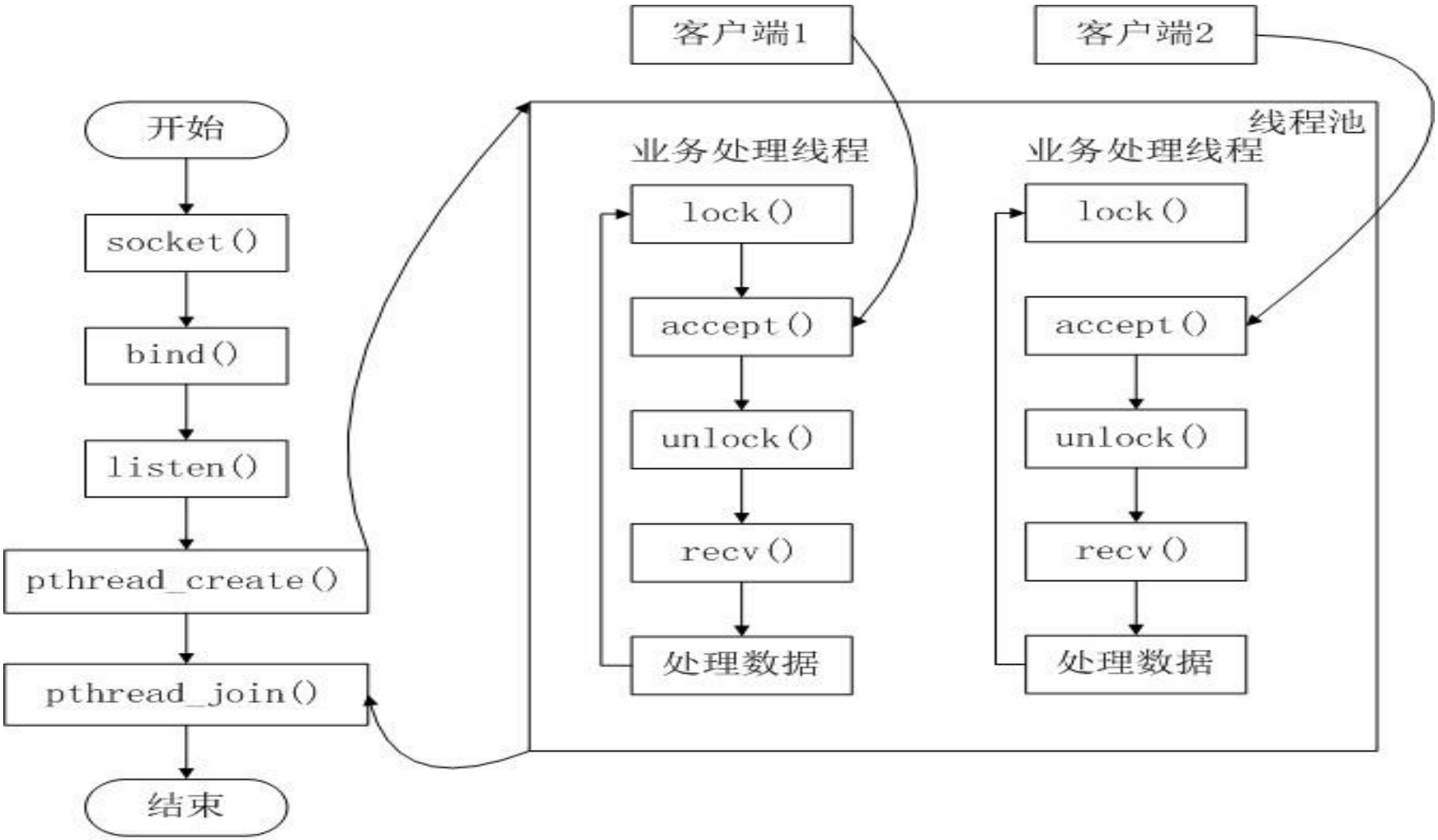
1.5 进程池——举例

```
static void handle_fork(int sockfd){
    ……
    while(1){
        len = sizeof(struct sockaddr);
        newfd = accept(sockfd,(struct sockaddr*) &c_addr, &len)
            ……
        len = read(newfd,buf,BUFLEN);
        if(len >0){
            ……
            now = time(NULL);
            ……
            send(newfd,buf,strlen(buf),0);
        }
      close(newfd);
    }
}
```

1.6 线程池

- 线程池
- 使用预先分配线程的并发服务器与之前使用预先分配进程的并发服务器的主要过程是一致的。主程序先建立多个处理线程，然后等待线程的结束，在多个线程中对客户端的请求进行处理。处理过程包括接收客户端的链接，处理数据，发送响应过程。这个可以结合线程池来进行相应的处理工作。

1.6 线程池——模型



并发服务器模型：预先分配多线程个数，使用互斥锁

1.6 线程池——举例

例5-13thread_poll.c

```
int main(int argc, char **argv)
{    ……
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
        ……
   bind(sockfd, (struct sockaddr*) &s_addr,sizeof(struct sockaddr)) ;
   listen(sockfd,listnum) ;
    ……
    for(i = 0; i < THREADNUM; i++){
      pthread_create(&thread_s[i],NULL,handle_thread,(void *)&sockfd);
    }
   for(i = 0; i < THREADNUM; i++){
        pthread_join(thread_s[i],NULL);
    }
    close(sockfd);
    return 0;
}
```

1.6 线程池——举例

```
static void *handle_thread(void *argv){
    ……
    while(1){
        len = sizeof(struct sockaddr);
        pthread_mutex_lock(&ALOCK);
        newfd = accept(sockfd,(struct sockaddr*) &c_addr, &len);
            ……
        pthread_mutex_unlock(&ALOCK);
        ……
        len = read(newfd,buf,BUFLEN);
        if(len >0){
            ……
            now = time(NULL);
            ……
            send(newfd,buf,strlen(buf),0);
        }close(newfd);    }
    return NULL;}
```