电子科技大学
University of Electronic Science and Technology of China

Lecture 3
Concurrent Servers

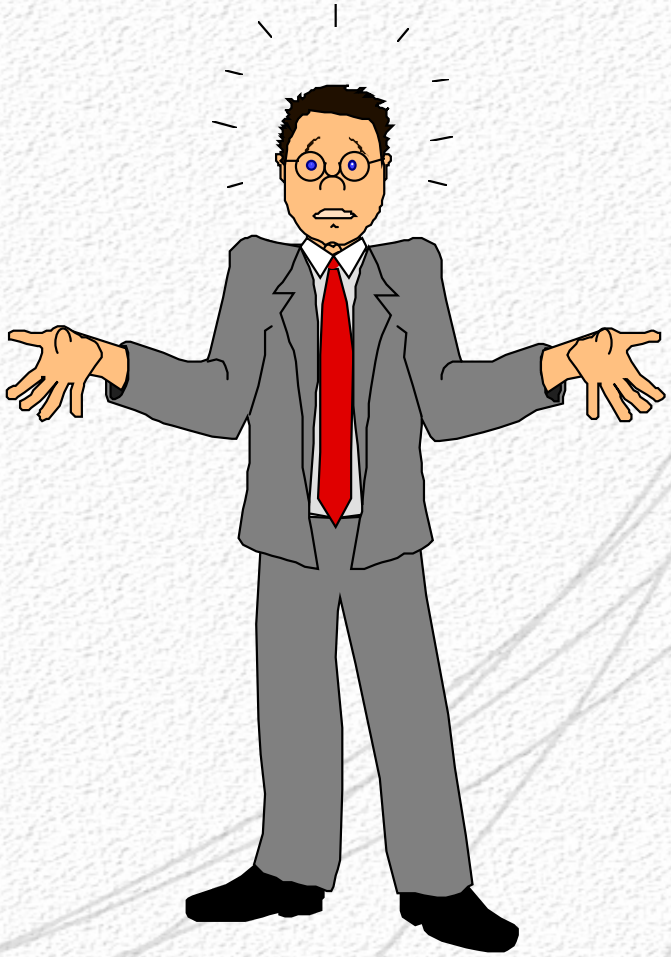
Ren Liyong

电子科技大学信软学院

www.uestc.edu.cn

目 录

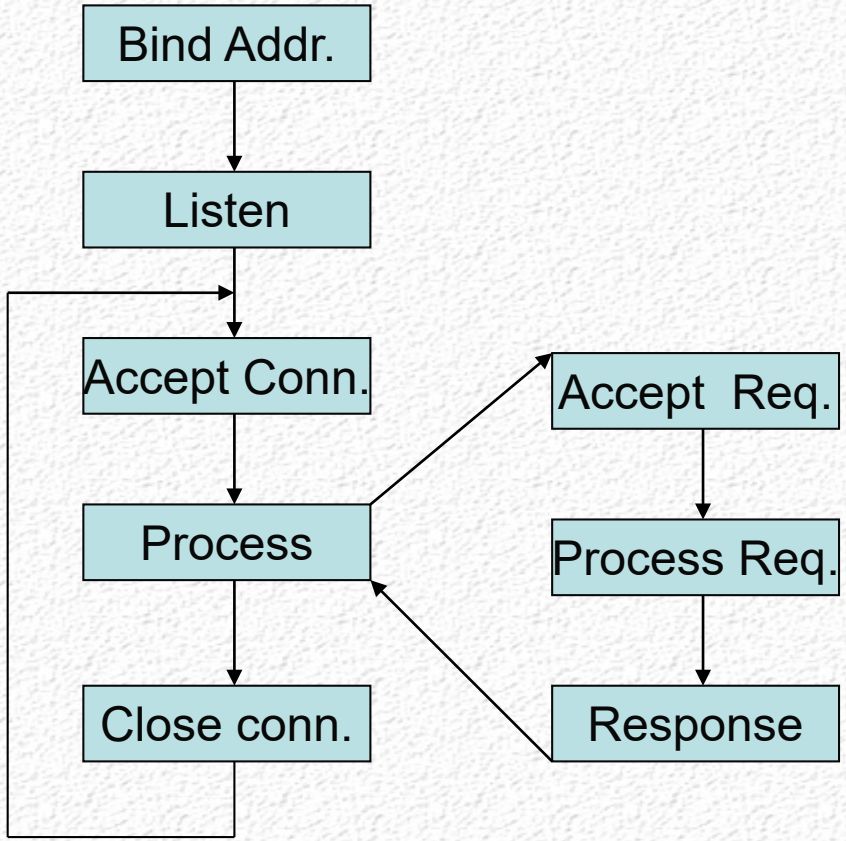
- 服务器分类技术
- 进程与线程
- 多进程服务器
- 多线程服务器



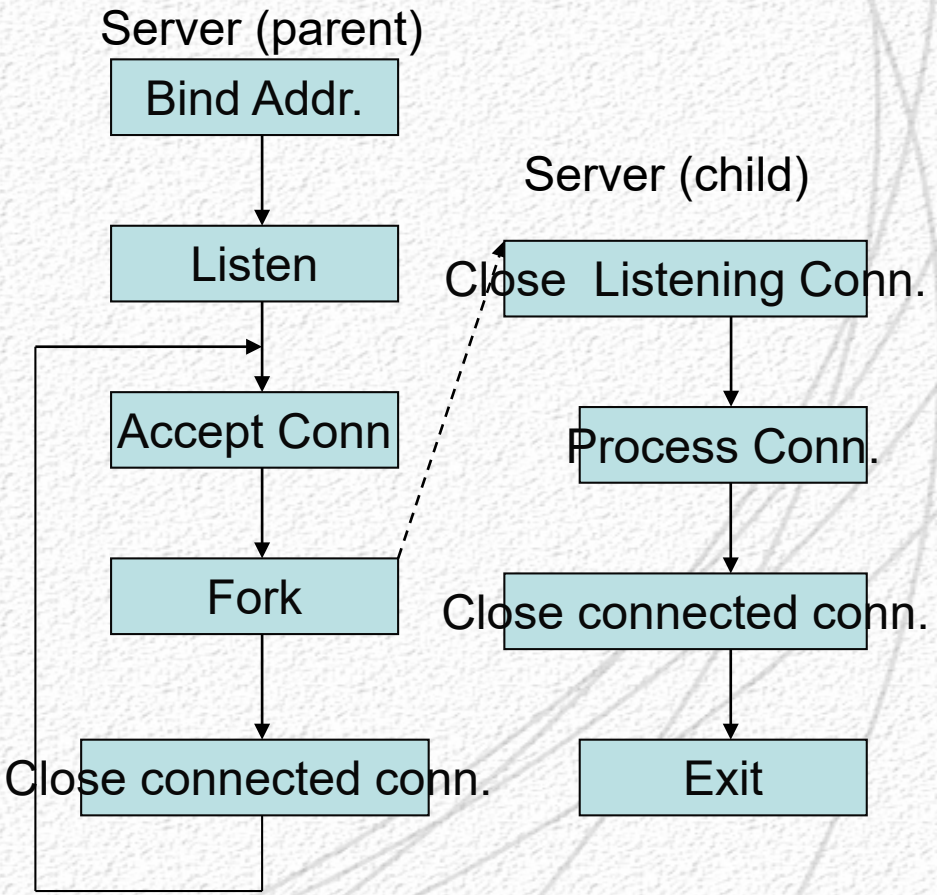
服务器分类

- **按连接类型分类**
  - 面向连接的服务器（如tcp）
  - 面向无连接的服务器（如udp）
- **按处理方式分类**
  - 迭代服务器
  - 并发服务器
- **按状态保存分类**
  - 有状态服务器
  - 无状态服务器

Iterative vs. Concurrent Server



TCP Iterative server



TCP Concurrent Server

“进程”基本概念

- 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。
- 进程间正常情况下，互不影响，一个进程的崩溃不会造成其他进程的崩溃。
- 当进程间共享某一资源时，需注意两个问题：同步问题和通信问题。

Create process

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void)
        returns: 0 in child, Process ID of child in parent, -1 on error.
```

- All descriptors open in the parent before the call to ***fork*** are shared with the child after ***fork*** returns.
- 非常重要的是：fork后，父子进程均需要将自己不使用的描述字关闭，有两方面的原因：（1）以免出现不同步的情况；（2）最后能正常关闭描述字

Create process （cont.）

- 子进程继承父进程的大部分属性，如：
  - 实际UID,GID和有效UID,GID；环境变量；UID、GID设置模式位；进程组号；控制终端；当前工作目录；根目录；文件创建掩码；文件长度限制；预定值（如优先级）等；
- 但子进程也有与父进程不同的属性，如：
  - 进程号；父进程号；子进程的用户时间和系统时间被初始化为0；子进程的超时时钟设置为0；子进程的信号处理函数指针置为空；子进程不继承父进程的记录锁等；

Terminate Process

```
#include <stdlib.h>
void exit(int status);
```

- 本函数终止调用进程。关闭所有子进程打开的描述符，释放占用的内存资源，并向父进程发送SIGCHLD信号。
- 进程退出前，需要刷新I/O缓冲区，并执行由atexit或on_exit注册的函数。

```
#include <unistd.h>
void _exit(int status);
```

- 该函数除不执行上述exit函数的第二条功能外，其他与exit函数完全一样。（一般情况下，fork的子进程调用_exit终止自己，而不是调用exit）。

获取子进程终止信息

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

返回：终止子进程的ID－成功；-1－出错；stat_loc存储子进程的返回值；

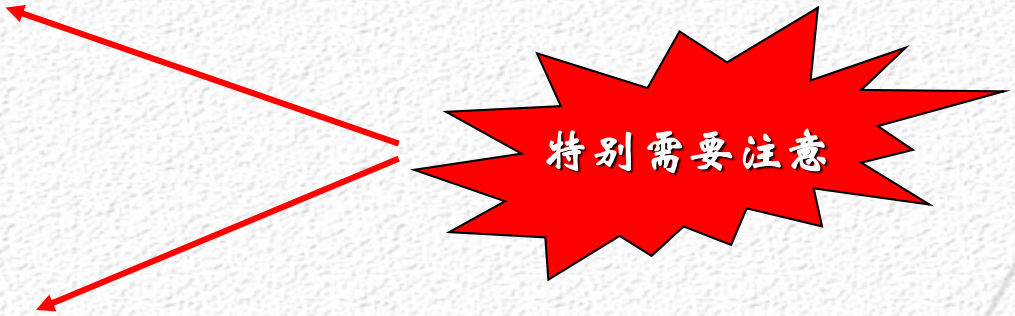
- 该函数将挂起当前进程，直到有一个子进程终止或者被信号中断。
- 当调用该系统调用时，如果有一个子进程已经终止，则该系统调用立即返回，并释放子进程所有资源。

多进程并发服务器模板

```
……
int main(void)
{
    int    listenfd, connfd;
    pid_t pid;
    int    BACKLOG = 5;
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)  {
          perror(“Create socket failed.”);
          exit(1);
    }
    bind(listenfd, …);
    listen(listenfd, BACKLOG);
    while(1) {
          if ((connfd = accept(sockfd, NULL, NULL)) == -1)  {
                perror(“Accept error.”);
                 exit(1);
          }
```

多进程并发服务器模板（cont.）

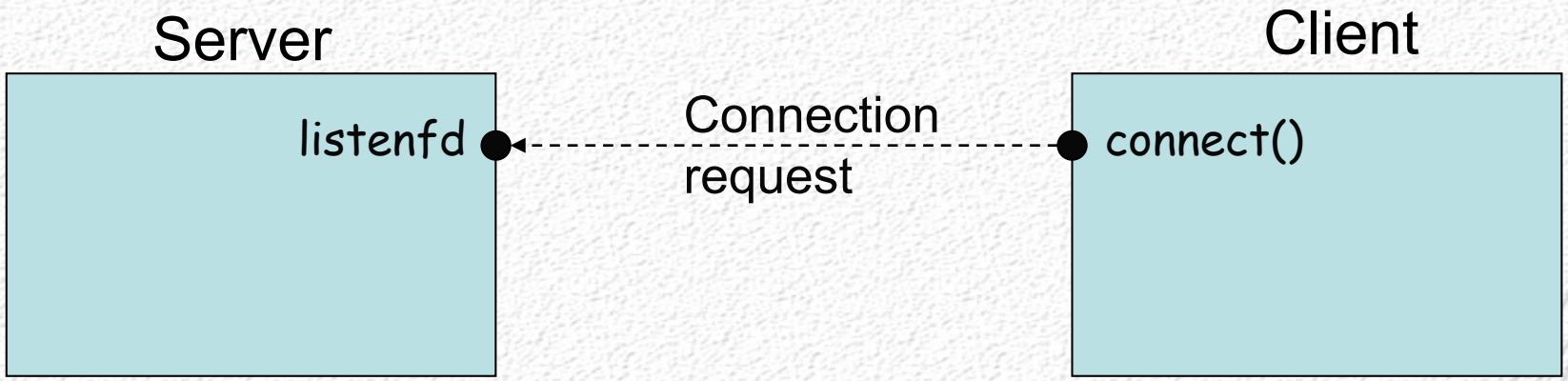
```
        if ((pid = fork()) >0) {
                close(connfd);
                ……
                conntinue;
        }
        else if (pid == 0)  {
                close(listenfd);
                ……
                close(connfd);
                _exit(0);
        }
        else {
                perror(“Create child process failed.”);
                exit(1);
        }
    }
}
```



一点说明

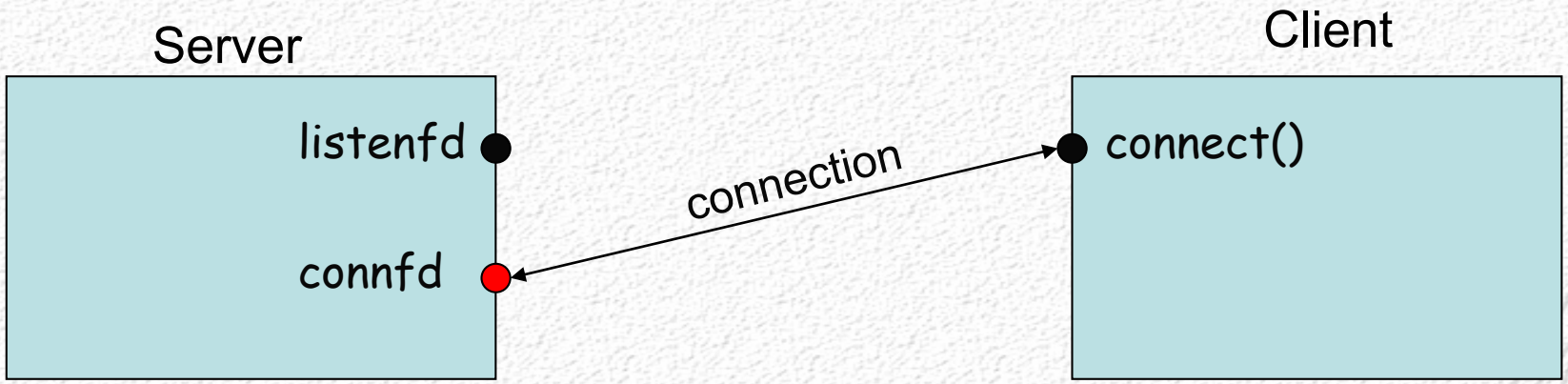
- 从以上模板看出，产生新的子进程后，父进程要关闭连接套接字，而子进程要关闭监听套接字，主要原因是：
  - 关闭不需要的套接字可节省系统资源，同时可避免父子进程共享这些套接字可能带来的不可预计的后果；
  - 另一个更重要的原因，是为了正确地关闭连接。和文件描述符一样，每个套接字描述符都有一个“引用计数”。当**fork**函数返回后，**listenfd**和**connfd**的引用计数变为**2**，而系统只有在某描述符的“引用计数”为**0**时，才真正关闭该描述符。

Status of concurrent server



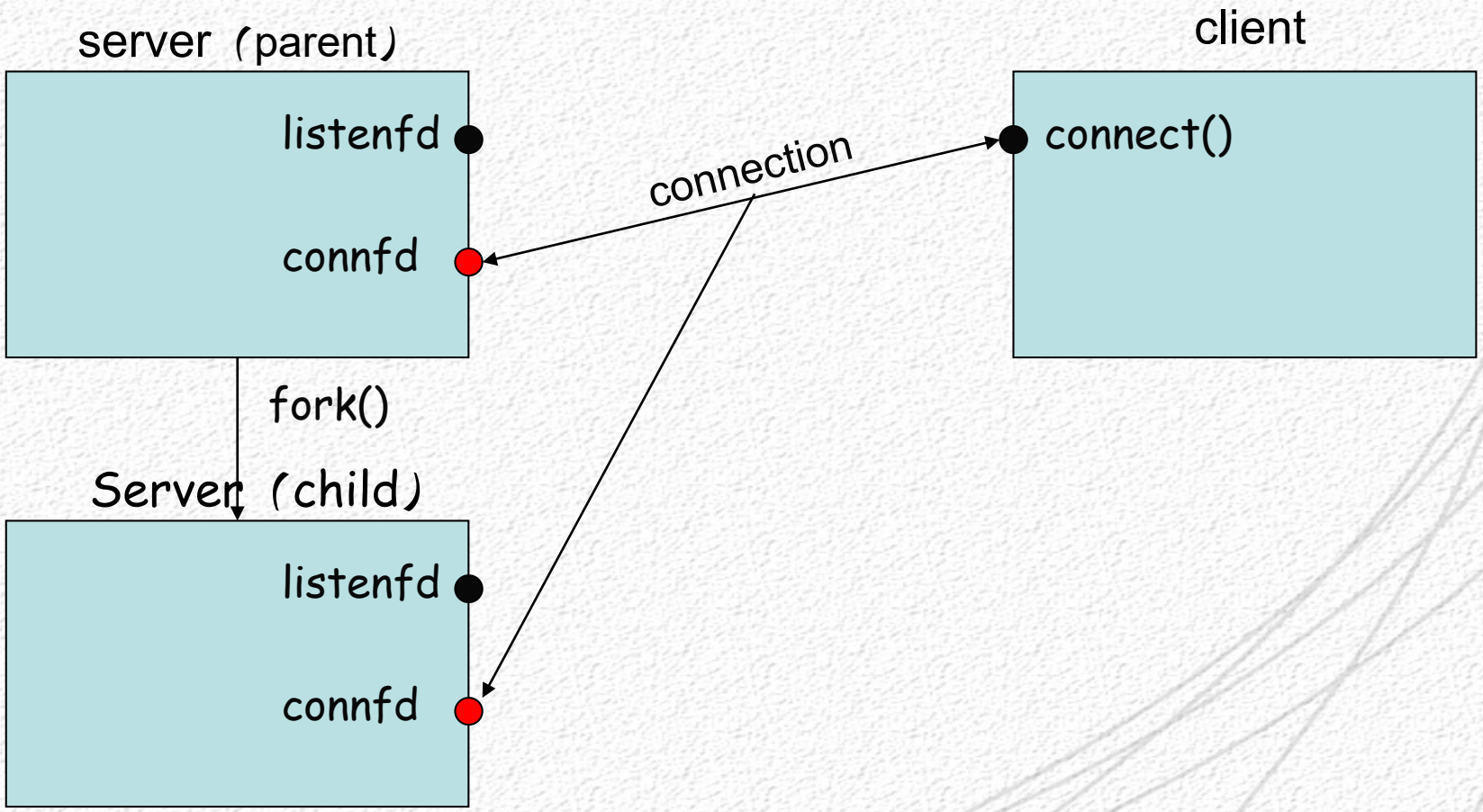
Status of client-server before call to ***accept***

Status of concurrent server （cont.）



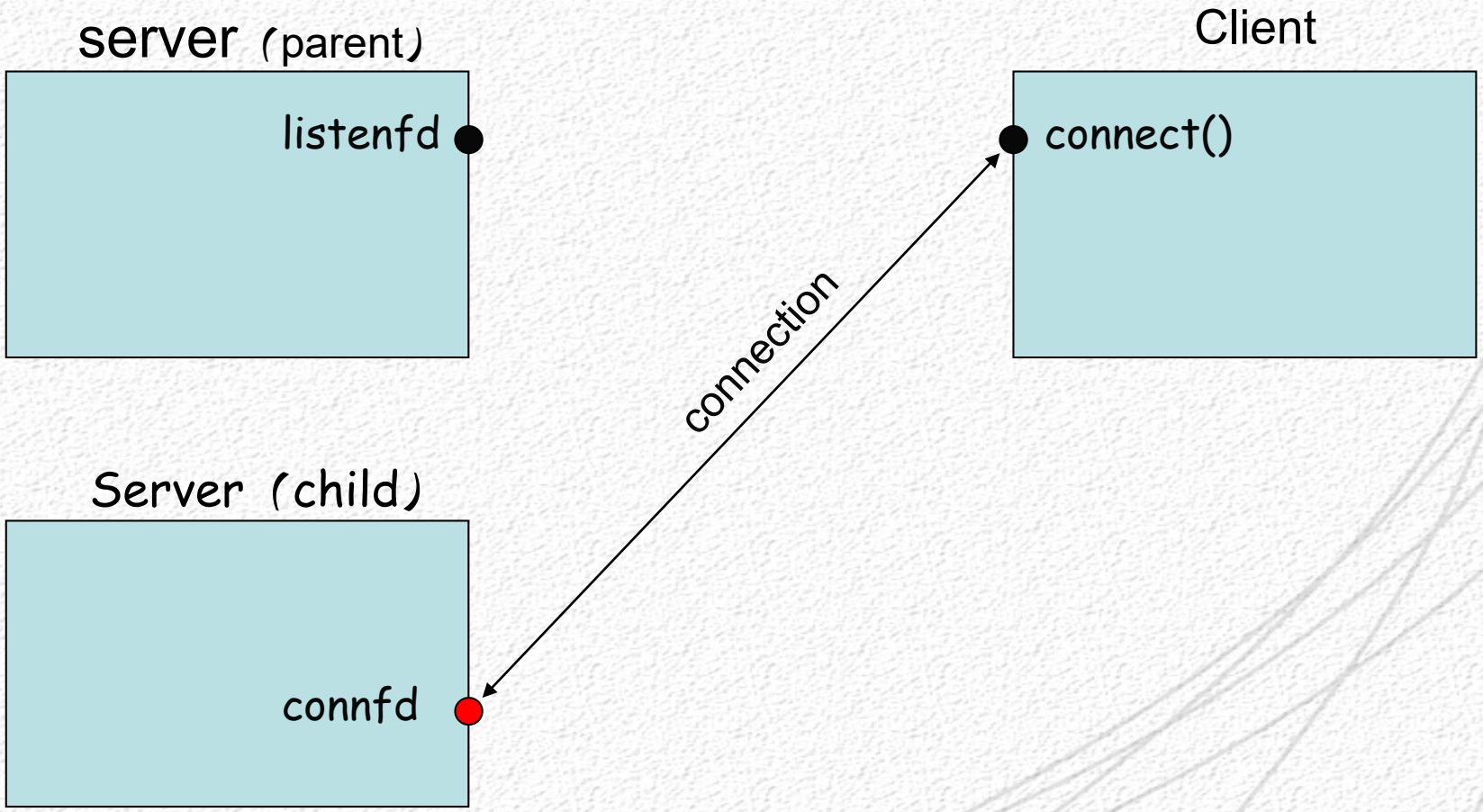
Status of client-server after return from ***accept***

Status of concurrent server （cont.）



Status of client-server after ***fork*** returns

Status of concurrent server （cont.）



Status of client-server after parent and child close appropriate sockets

Exam. Multi-process Concurrent server

- 该实例包括服务器程序和客户程序，具体功能如下：
  - 服务器等待客户连接，连接成功后显示客户地址，接着接收该客户的名字并显示，然后接收来自客户的信息（字符串），将该字符串反转，并将结果送回客户。要求服务器具有同时处理多个客户的能力。
  - 客户首先与服务器连接，接着接收用户输入客户的名字，将该名字发送给服务器，然后接收用户输入的字符串，并发送给服务器，然后接收服务器返回的经处理后的字符串，并显示之。当用户输入**Ctrl+D**，终止连接并退出。

多进程并发服务器—服务器

```
#include <stdio.h>
……
#define  PORT      1234
#define  BACKLOG           2
#define MAXDATASIZE      1000
void process_cli(int connectfd, sockaddr_in client);

int main(void)
{
    int     listenfd, connectfd;
    pid_t pid;
    struct sockaddr_int      server, client;
    int     sin_size;

    /* Create TCP Socket */
    ……
```

多进程并发服务器—服务器

```
/* Bind server address to listenfd. */
……
/* Listen on listenfd */
……
sin_size = sizeof(struct sockaddr_int);
while(1) {
        if ((connectfd = accept(listenfd, (struct sockaddr *)&client, &sin_size)) == -1) {
                perror("accept error."); exit(1); }
        if ((pid = fork()) > 0) {
                close(connectfd);
                continue;
        }
        else if (pid == 0) {
                close(listenfd);
                process_cli(connectfd, client);
                exit(1);
        }
```

多进程并发服务器—服务器

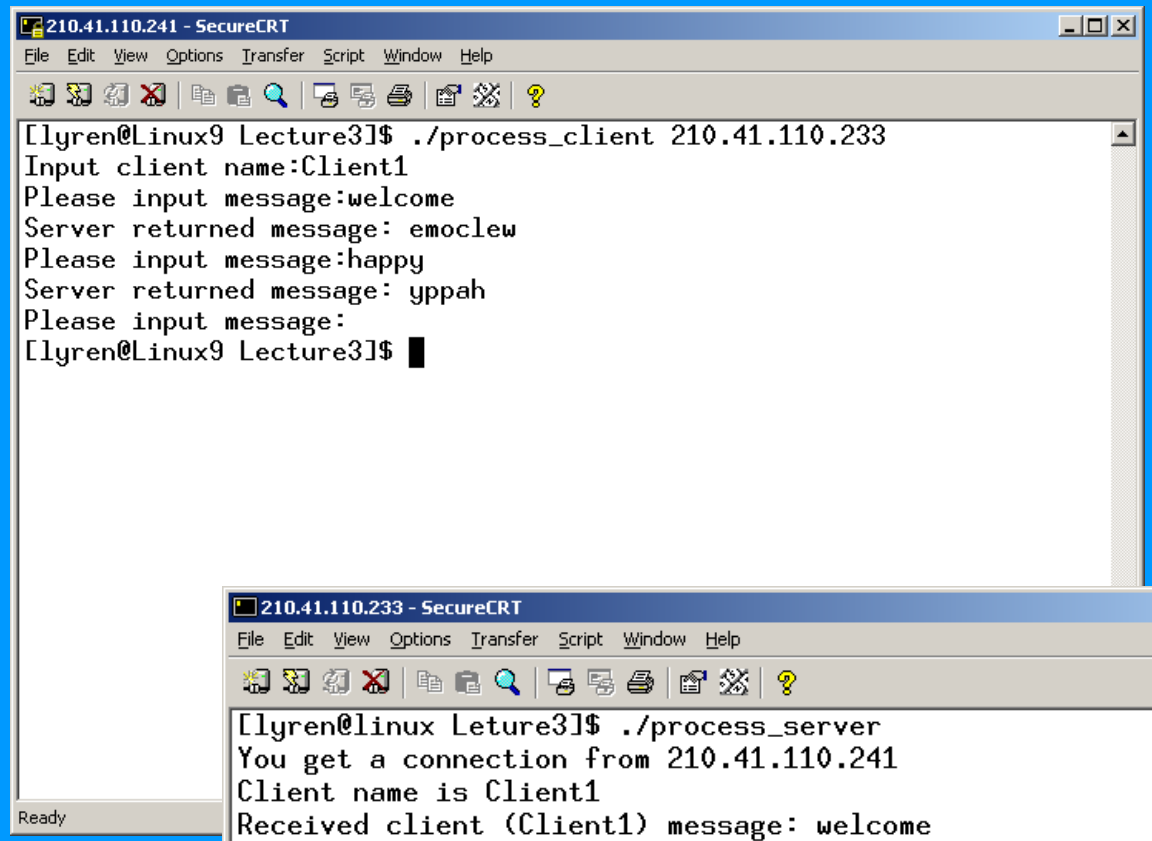
```
            else {
                    printf("fork error.\n");
                    exit(0);
            }
    }
    close(listenfd);
}
```

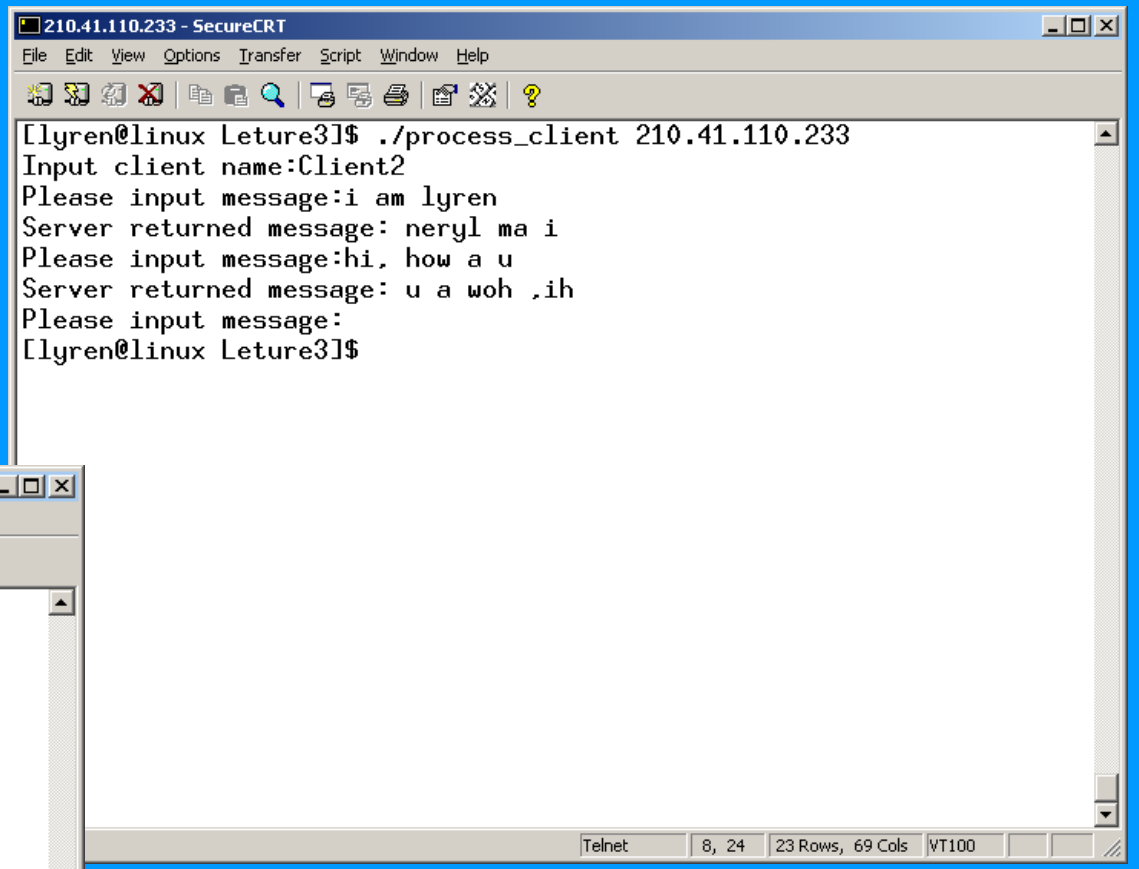
多进程并发服务器—服务器

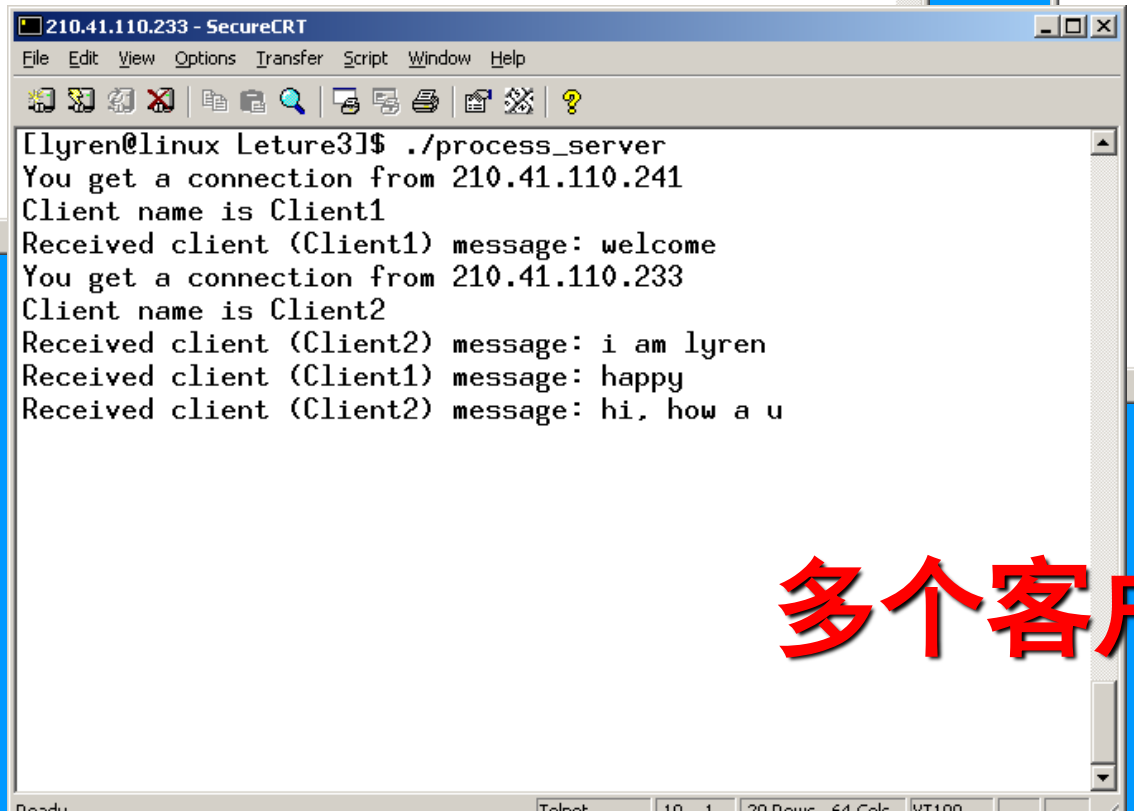
```
void process_cli(int connectfd, sockadd_in client) {
    int     num;
    char recvbuf[MAXDATASIZE], sendbuf[MAXDATASIZE], cli_name[MAXDATASIZE];
    printf("You got a connection from %s.\n", inet_ntoa(client.sin_addr));
    num = recv(connectfd, cli_name, MAXDATASIZE, 0);
    if (num == 0) {
            close(connectfd);
            printf("cllient disconnected.\n");
            return;
    }
    cli_name[num-1] = '\0';
    printf("Client name is %s.\n",cli_name);:
```

多进程并发服务器—服务器

```
    while (num = recv(connectfd, recvbuf, MAXDATASIZE,0) {
            recvbuf[num] = '\0';
            printf("Received client (%s) message: %s\n", cli_name, recvbuf);
            for(int i=0; i < num – 1; i++)
                    sendbuf[i] = recvbuf[num-i-2];
            sendbuf[num-1] = '\0';
            send(connectfd, sendbuf, strlen(sendbuf), 0);
    }
    close(connectfd);
}
```


210.41.110.241 - SecureCRT
File Edit View Options Transfer Script Window Help
[lyren@Linux9 Lecture3]$ ./process_client 210.41.110.233
Input client name:Client1
Please input message:welcome
Server returned message: emoclew
Please input message:happy
Server returned message: yppah
Please input message:
[lyren@Linux9 Lecture3]$
Ready


210.41.110.233 - SecureCRT
File Edit View Options Transfer Script Window Help
[lyren@linux Leture3]$ ./process_client 210.41.110.233
Input client name:Client2
Please input message:i am lyren
Server returned message: neryl ma i
Please input message:hi, how a u
Server returned message: u a woh ,ih
Please input message:
[lyren@linux Leture3]$
Telnet 8, 24 23 Rows, 69 Cols VT100


210.41.110.233 - SecureCRT
File Edit View Options Transfer Script Window Help
[lyren@linux Leture3]$ ./process_server
You get a connection from 210.41.110.241
Client name is Client1
Received client (Client1) message: welcome
You get a connection from 210.41.110.233
Client name is Client2
Received client (Client2) message: i am lyren
Received client (Client1) message: happy
Received client (Client2) message: hi, how a u

多个客户同时请求处理

The problems of Multi-process server

- In the traditional Unix model, when a process need something performed by another entity it ***forks*** a child process and lets the child performs the processing. There are **problems** with ***fork***:
  - ***Fork*** is **expensive**.
  - Inter-process communication is required to pass information between the parent and child after the ***fork***.
  - The numbers of concurrent process is **limits**.

“线程”基本概念

- Threads help with both problems. Thread are sometimes called *lightweight processes*. That is, thread creation can be 10~100 times faster than process creation.
- All threads within a process share the same global memory and following:

Shared	private
Process instuctions	thread ID
Most data	set of registers
Open files	stack
Signal handlers	errno
User and group ID	signal mask

Thread Functions（1）

```
#include <pthread.h>
int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*func)(void *), void
    *arg);
```

returns: -0 if OK, positive Exxx value on error.

- When a program is started by ***exec*** , a single thread is created, called the ***initial thread*** or ***main thread.*** **Additional threads** are created by ***pthread_create.***
- Notice the declarations of ***fun*** and ***arg***. The function takes on argument, a generic pointer (void *) , and return a same one.

Thread Functions （2）

```
#include <pthread.h>
void pthread_exit(void *status);
```

Does not return to caller

- One way for a thread to terminate is to call this function.
- There are tow other way for a thread to terminate:
  - The function that started the thread can return.
  - If the ***main*** function of the process return or if any thread calls exit, the process terminates, including any threads.

多线程并发服务器模板

```
……
void *start_routine( void *arg);
int main(void)  {
    int    listenfd, connfd;
    pthread_t      tid;
    type               arg;
    /* Create TCP socket */
    ……
    /* Bind socket to address */
    ……
    /* Listen */
    ……
    while(1) {
        /* Accept connection */
        if ((pthread_create(&tid, NULL, start_routine, (void *)&arg))
            /* handle exception */
        ……
    }
    ……
}
```

给新线程传递参数

- 由于同一个进程内的所有线程共享内存和变量，因此在传递参数时需作特殊处理，下面参考如下几种方法：
  - 传递参数的普通方法
  - 通过指针传递参数
  - 通过分配arg的空间来传递参数
  - 还可以通过加锁等同步设施来实现传递参数；

传递参数的普通方法

- 由于线程创建函数只允许传递一个参数，因此当需要传递多个数据时，应首先将这些数据封装在一个结构中。

```
void *start_routine(void *arg);
struct ARG {
    int connfd;
    int other;
}
int main() {
    …
    ARG arg;
    …
    pthread_create(&tid, NULL, start_routine, (void *)&arg);
    …
}
```

传递参数的普通方法（续）

```
void *start_routine(void *arg) {
    ARG info;
    info.connfd = ((ARG *)arg) -> connfd;
    info.other = ((ARG *)arg) -> other;
    …
}
```

- 这种方法有问题，对一个客户可以工作，但多个客户则可能出现问题。（请考虑：出现什么问题？）

通过指针传递参数

- 这种方法首先将要传递的数据转换成通用指针类型，然后传递给新线程，新线程再将其还原成原数据类型：

```
void *start_routine(void *arg);
int main(void) {
    int connfd;
    …
    pthread_create(&tid, NULL, start_routine, (void *)connfd);
    …
}
void *start_routine(void *arg) {
    int connfd;
    connfd =(int )*arg;
    …
}
```

- 这种方法虽然简单，但却有**很大的局限性**。如：要求arg的类型必须能被正确地转换成通用指针类型，这就要求arg的长度必须小于或等于通用指针类型的长度。从而导致可传递的参数很少。

通过分配arg的空间来传递

- 主线程首先为每个新线程分配存储arg的空间，再将arg传递给新线程使用，新线程使用完后要释放该空间。

```
void *start_routine(void *arg);
int main(void) {
    ARG * arg;
    int connfd;
    loop {
            …
            arg = new ARG;
            arg -> connfd = connfd;
            pthread_create(&tid, NULL, start_routine, (void *)arg);
            …
    }
}
```

通过分配arg的空间来传递（续）

```
void *start_routine(void *arg) {
    ARG info;
    info.connfd = ((ARG *)arg) ->connfd;
    …
    delete arg;
}
```

多线程并发服务器

```
#include <stdio.h>
……
#define  PORT       1234
#define  BACKLOG            2
#define MAXDATASIZE        1000
void process_cli(int connectfd, sockaddr_in client);
void *start_routine(void *arg);
struct ARG {
    int                connfd;
    sockaddr_in    client;
};

int main(void)
{
    int     listenfd, connectfd;
    pthread_t        tid;
    ARG  *arg;
```

多线程并发服务器（cont.）

```
    struct sockaddr_in     server, client;
    int sin_size;
    /* Create TCP Socket */   …
    /* Bind server address to listenfd. */ …
    /* Listen on listenfd */ …
    sin_size = sizeof(struct sockaddr_in);
    while(1) {
          if ((connectfd = accept(listenfd, (struct sockaddr *)&client, &sin_size))       -1)
                /* handle error */
          arg = new ARG;
          arg -> connfd = connectfd;
          memcpy((void *)&arg -> client, &client, sizeof(client));

          if (pthread_create(&tid, NULL, start_routine, (void *)arg)
                /* handle erroe */
    }
    close(listendfd);
}
```

这是拷贝结构的常用方法。

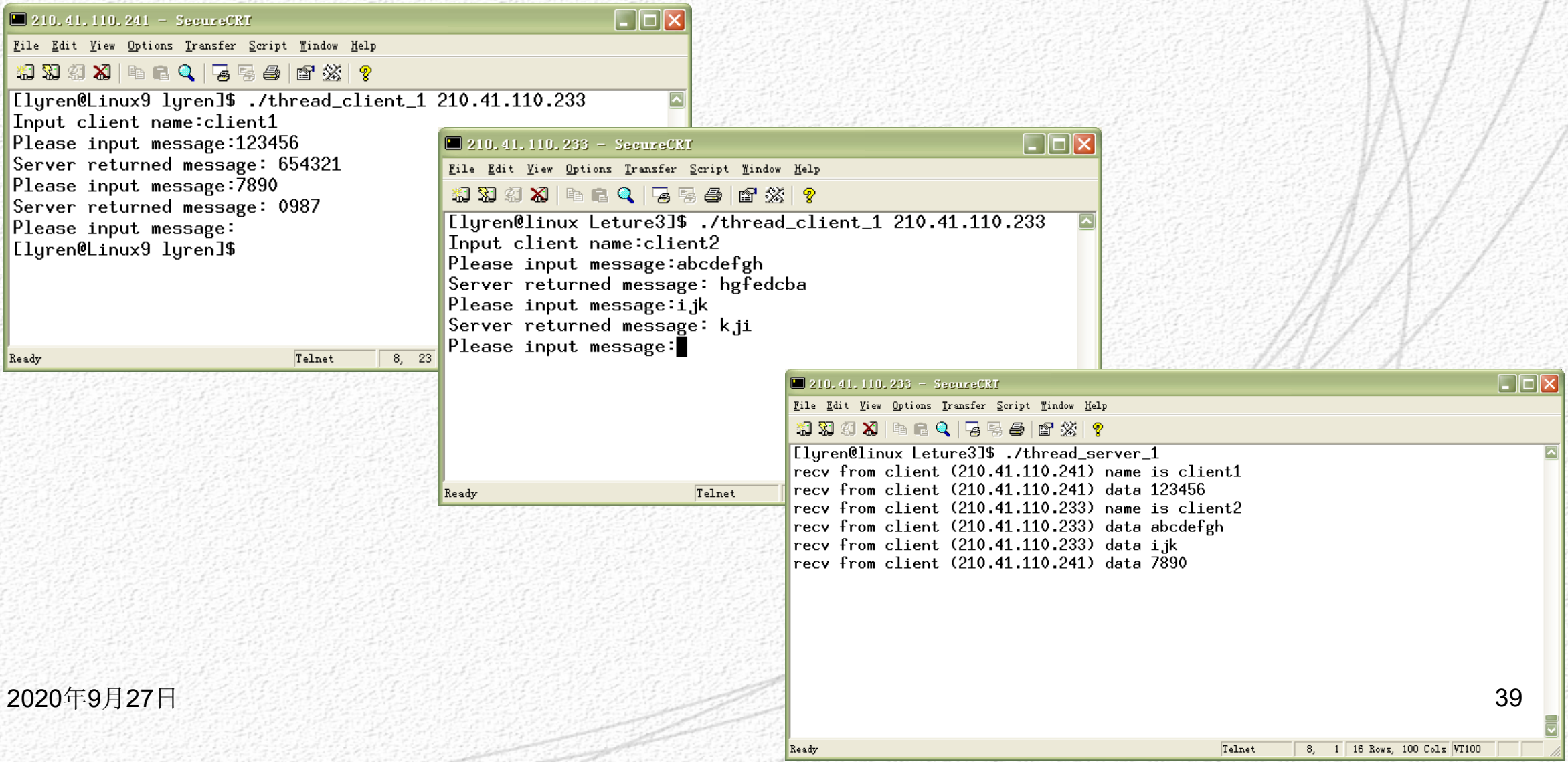
多线程并发服务器（cont.）

```
void process_cli(int connectfd, sockaddr_in client) {
    …
}
void *start_routine(void *arg) {
    ARG *info;
    info = (ARG *)arg;
    process_cli(info -> connfd, info -> client);
    delete arg;
    pthread_exit(NULL);
}
```

多线程并发服务器（cont.）

- 运行结果（略）：该程序能正常工作。
- **需要注意的是**：与多进程并发服务器不同的是，由于多个线程间共享相同的内存空间和描述字，因此pthead_create后，不能关闭监听套接字和连接套接字，否则程序不能正常工作。（可以做试验验证）

多线程并发服务器（运行结果）



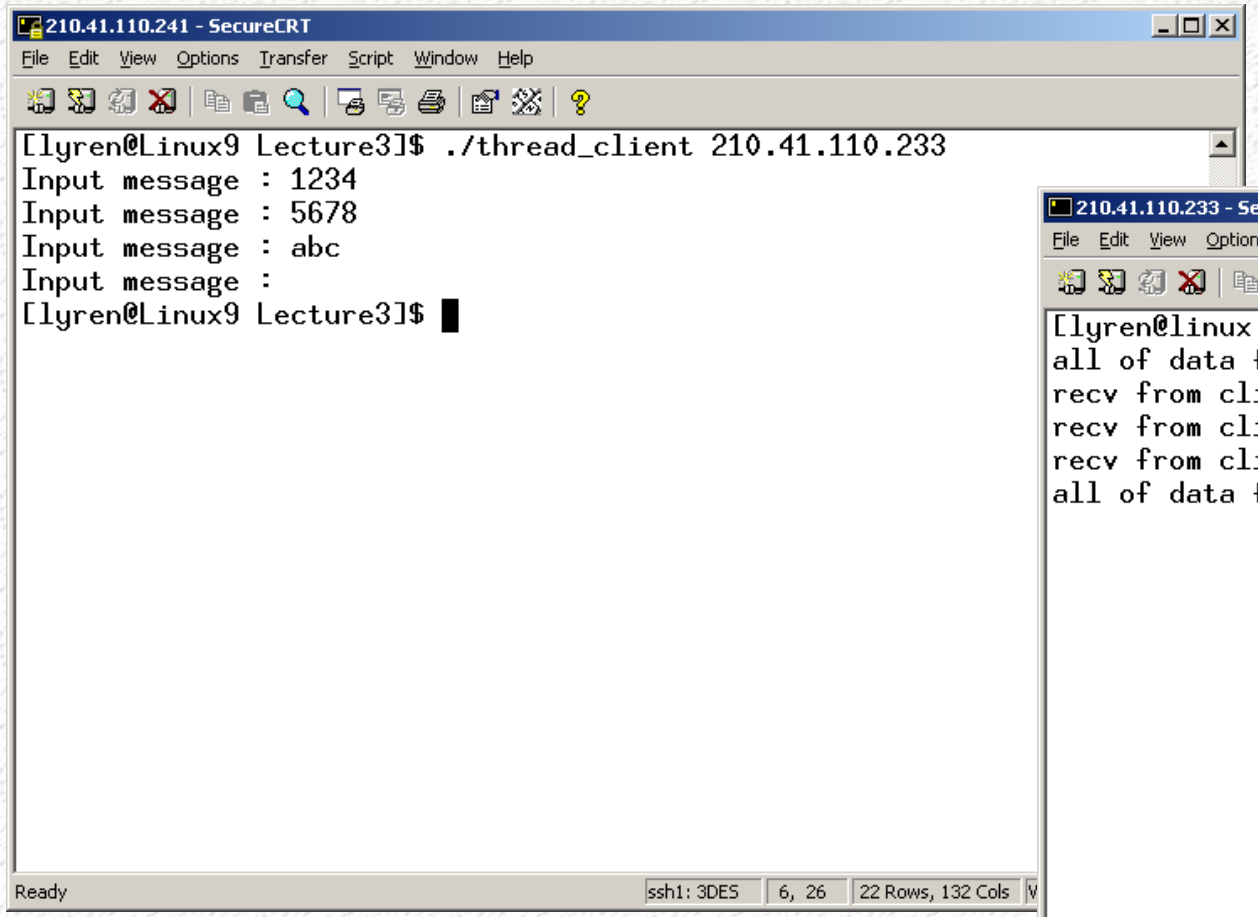
线程安全问题

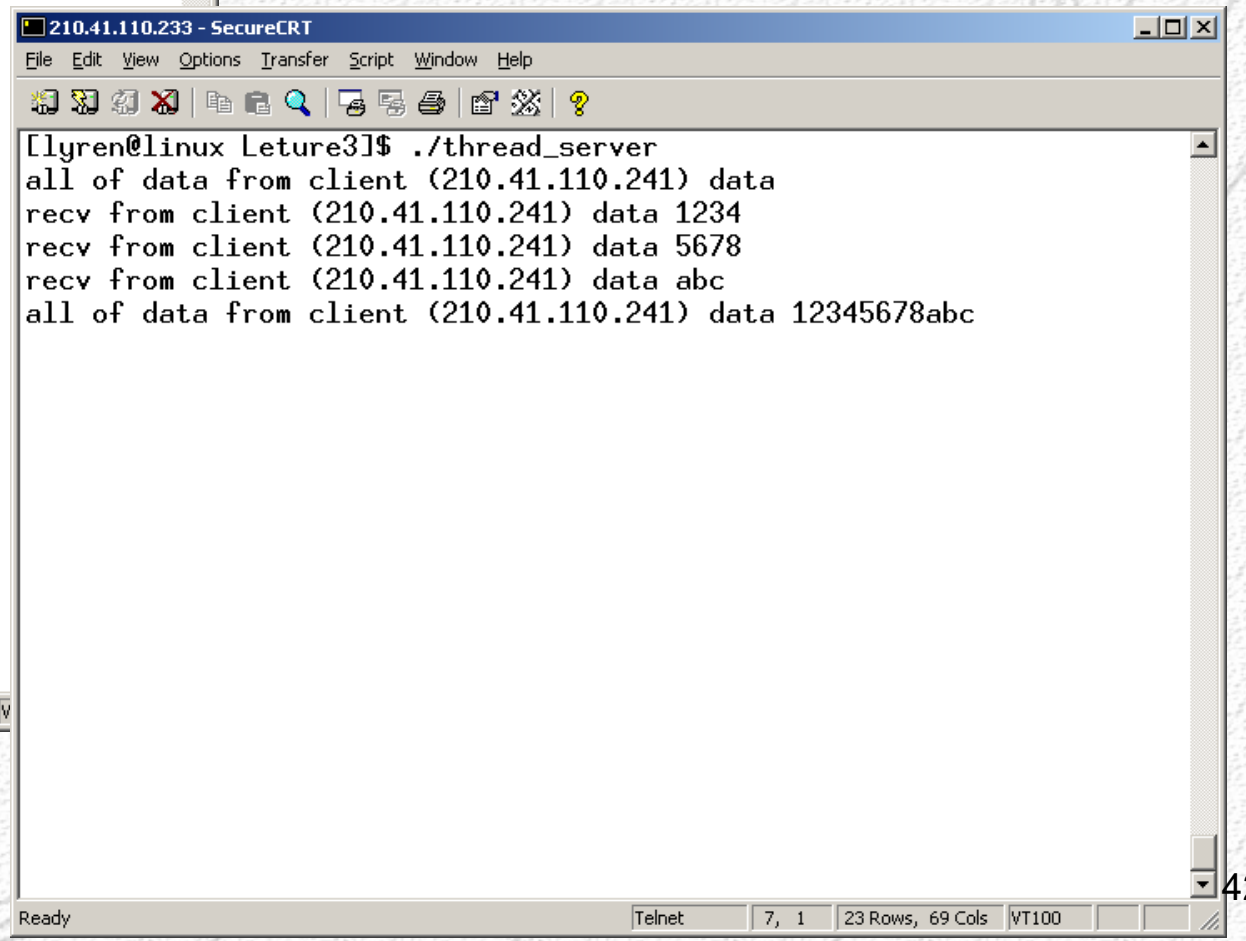
- 线程安全问题是一个非常复杂的问题。简单地说，就是多个线程在操作共享数据时出现的混乱情况，这种情况可能导致不可预测的后果。
- 解决线程安全问题的方法主要有两种：一是使用线程安全函数：posix定义的以”_r”结尾的函数，二是使用线程专用数据(TSD)。
- 下面举例说明线程安全问题

线程安全问题（cont.）

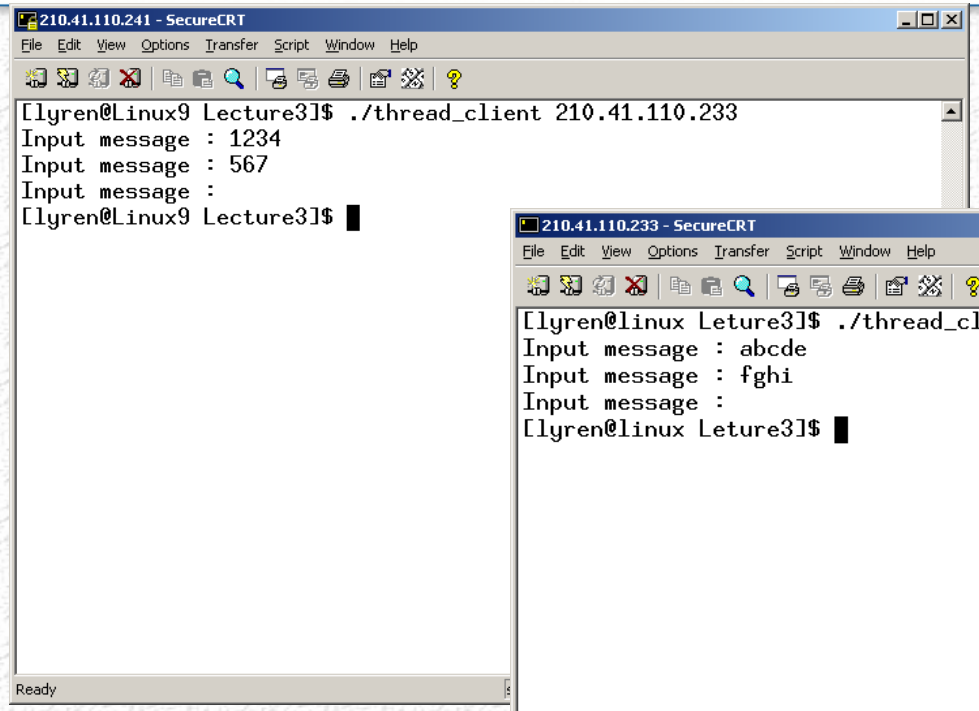
```
void process_cli(int connectfd, sockaddr_in client) {
    int num;
    char cli_data[5000], recvbuf[MAXDATASIZE];
    …
    while (num = recv (connectfd, recvbuf, MAXDATASIZE, 0)) {
        savedata( recvbuf, num, cli_data);
        …
    }
}

void savedata(char *recvbuf, int len, char *cli_data) {
    static int index = 0;
    for (int i=0; i< len -1; i++)
        cli_data[index++] = recv[i];
    cli_data[index] = '\0';
}
```

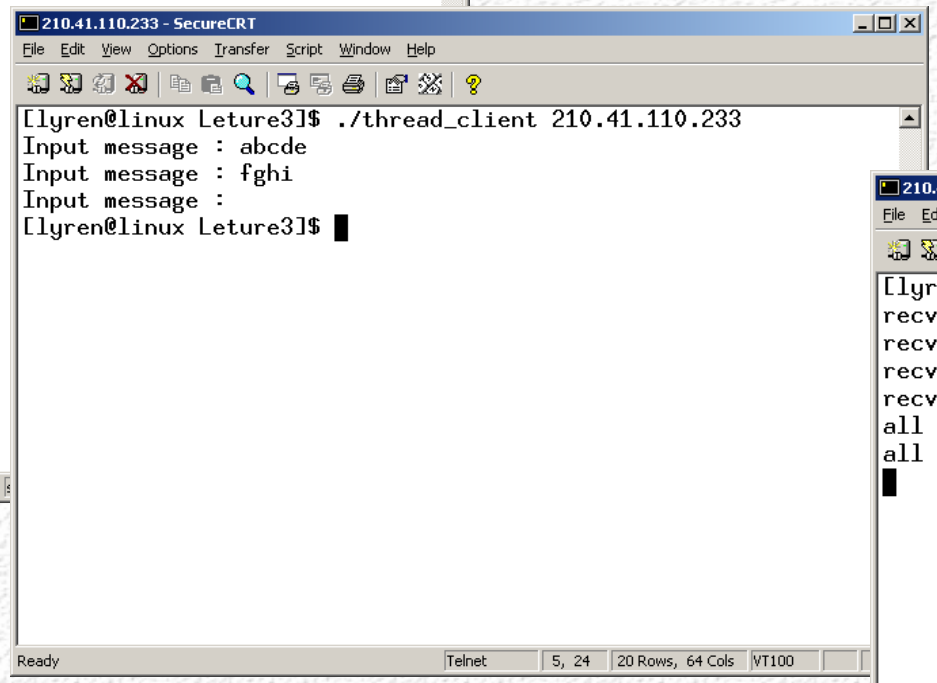
单个客户的情况

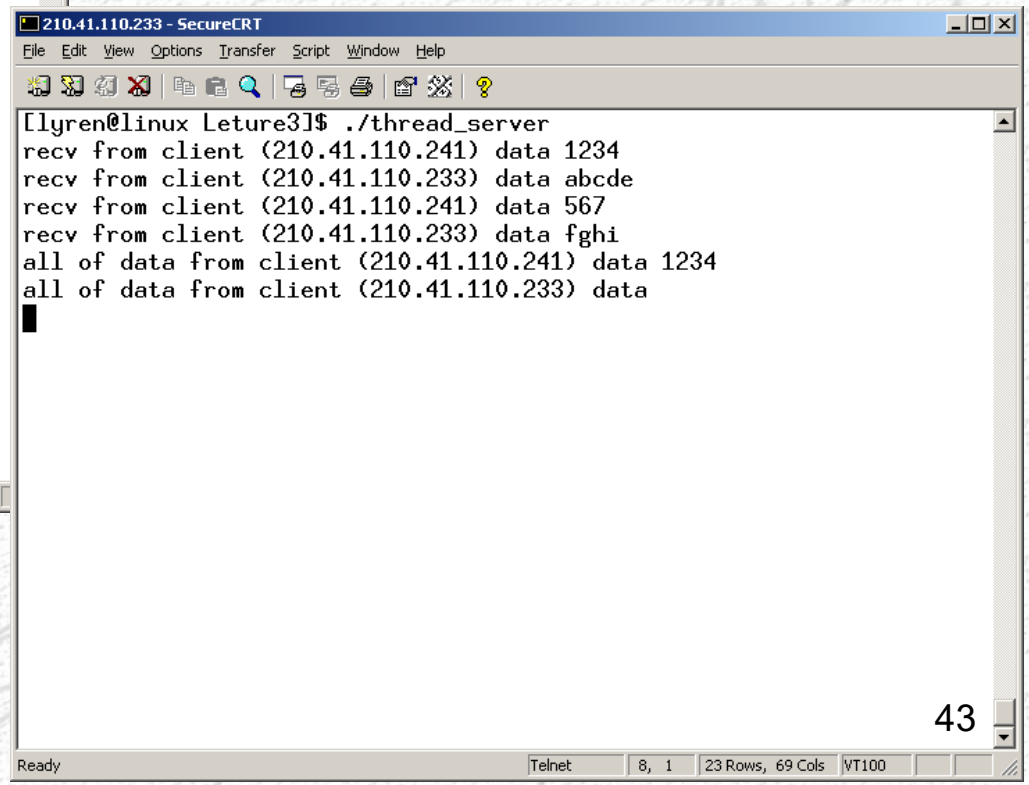

210.41.110.241 - SecureCRT
File Edit View Options Transfer Script Window Help
[lyren@Linux9 Lecture3]$ ./thread_client 210.41.110.233
Input message : 1234
Input message : 5678
Input message : abc
Input message :
[lyren@Linux9 Lecture3]$
Ready
ssh1: 3DES
6, 26
22 Rows, 132 Cols


210.41.110.233 - SecureCRT
File Edit View Options Transfer Script Window Help
[lyren@linux Leture3]$ ./thread_server
all of data from client (210.41.110.241) data
recv from client (210.41.110.241) data 1234
recv from client (210.41.110.241) data 5678
recv from client (210.41.110.241) data abc
all of data from client (210.41.110.241) data 12345678abc
Ready
Telnet
7, 1
23 Rows, 69 Cols
VT100

两个客户交叉请求的情况







Thread-Specific Data (TSD)

- This is a common problem when converting existing functions to run in a threads environment and there are various solutions:
  - User thread-specific data
  - Change the calling sequence so that the caller packages all the arguments into a structure.

第三次作业

- 利用多线程技术实现如下并发网络程序，要求对上课时的实现进行完善，利用线程专用数据TSD实现。服务器和客户程序分别实现如下功能：

1、服务器等待客户连接，连接成功后显示客户地址，接着接收该客户的名字并显示，然后接收来自客户的信息（字符串），将该字符串反转，并将结果送回客户。要求服务器具有同时处理多个客户的能力。当某个客户断开连接时，打印所有该客户输入的数据。

2、客户首先与服务器连接，接着接收用户输入客户的名字，将该名字发送给服务器，然后接收用户输入的字符串，并发送给服务器，然后接收服务器返回的经处理后的字符串，并显示之。当用户输入Ctrl+D，终止连接并退出。