

计网第三次实验

姓名：李哲彦

学号：37220222203675

学院：信息学院

专业：计算机科学与技术

日期：2024年10月24日

实验目的

- 理解TCP和UDP协议主要特点
- 掌握socket的基本概念和工作原理，编程实现socket通信

实验内容与成果

任务1+任务2

任务1：完善socket客户端

任务要求

按以下要求，修改范例client_example.c，实现类似telnet连接echo服务器的效果：

- 为所有socket函数调用添加错误处理代码；
- 范例中服务器地址和端口是固定值，请将它们改成 允许用户以命令行参数形式输入；
- 范例中客户端发送的是固定文本“Hello Network!”， 请改成允许用户输入字符串，按回车发送；
- 实现循环，直至客户端输入“bye”退出。

代码实现

1、**添加错误处理信息：**各个 socket 函数在运行时均会有返回值，如果其返回值为 -1，则说明其运行出错，同时会直接设置 `errno` 参数。因此，每次运行相关函数时，检测其值是否为 -1 即可判断是否出错。若出错，则利用 `perror` 输出错误信息。

以连接服务器为例：

```
/* 连接服务器 */  
int tmp = connect(client_sock, (struct sockaddr *)&server_addr,
```

```

sizeof(server_addr));
    if(tmp == -1){
        perror("Connect Error!\n");
        return 0;
    }

```

2、**允许命令行输入：**在 main 函数中利用 argc 以及 argv 参数即可获取命令行输入的参数。首先判断参数输入是否正确。不正确则提醒用户输入。在这里，我设置为第一个参数为服务器 ip 地址，第二个参数为服务器端口号。

```

if (argc != 3) {
    printf("使用方法: %s <服务器IP地址> <服务器端口号>\n", argv[0]);
    return 0;
}

```

之后，通过 argv 参数获取命令行输入的参数。注意 argv 参数返回的均为字符串。需要进行转换。然后，在指定服务器地址时，按照输入的参数进行赋值即可。

```

/*获取端口号，转化为整数*/
int port_num = char_to_num(argv[2]);
char *ip_addr = argv[1];
printf("ip addr: %s    port number: %d\n", ip_addr, port_num);

/* 指定服务器地址 */
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(port_num);
inet_aton(ip_addr, &server_addr.sin_addr);
memset(server_addr.sin_zero, 0, sizeof(server_addr.sin_zero)); //零填充

```

3、**循环输入与 bye 退出：**在原有基础上，为实现循环输入并判断用户输入 bye 后退出，可以使用 while 循环结构。同时利用传送的字符串信息来通知 server 端。结构类似如下伪代码：

```

while(1):
    user input to str
    send_to_server(str) // 利用 str 来通知服务端
    if str == "bye":
        break
    receive_from_server(rec_str)

```

4、**Ctrl+C退出：**利用 signal 函数即可。在退出的时候需要发送 "bye"，然后关闭socket连接。

5、**服务端修改：**为实现循环输入，服务端其实也需要进行修改。可以利用与用户端类似的代码结构，用 while 循环结构不断接受 client 端传来的字符信息。如果为 "bye" 字符串，则服务端同

步退出。

运行结果

1、编译运行样例代码后，可以发现，客户机会发送 Hello World! 这一字符串。然后服务器会返回同样的字符串给客户机。

下图为客户端与服务端的运行情况。

```
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./server_example
Recv: Hello Network!
Send: Hello Network!
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ S

lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./client_example
Send: Hello Network!
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./client_example
Send: Hello Network!
Recv: Hello Network!
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$
```

2、接下来，对 client_example.c 按照要求进行改造。

可以看到运行结果，用户侧可以和服务侧进行多次问答。当用户侧输入 bye之后，两边都会结束程序。

```
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./server
Recv: abc
Send: abc
Recv: 1234
Send: 1234
Recv: abcdefg
Send: abcdefg
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$

lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./client
使用方法: ./client <服务器IP地址> <服务器端口号>
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./client 127.0.0.1 12345
Socket created successfully.
ip addr: 127.0.0.1 port number: 12345
Input your messages: abc
Send: abc
Recv: abc
Input your messages: 1234
Send: 1234
Recv: 1234
Input your messages: abcdefg
Send: abcdefg
Recv: abcdefg
Input your messages: bye
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ S
```

任务2：年份生肖查询服务器（TCP迭代）

任务要求：

按以下要求，修改范例server_example.c：

1. 为所有socket函数调用添加错误处理代码；
2. 范例中服务器地址和端口是固定值，请将它们改成允许用户 通过命令行参数形式输入；
3. 实现循环，直至客户机输入“bye”退出；
4. 服务器迭代地处理客户机请求：查询给定的testlist，将结果回复客户机；一个客户机退出后、继续接受下一个，按Ctrl+C可以终止服务器程序。

代码实现

- 1、**添加错误处理代码以及命令参数输入：**与 client 端的修改方法其实完全相同，继续利用 perror 函数以及 argc/argv 参数即可。在此不再赘述。
- 2、**实现循环：**其实为了实现任务1，任务2的该项内容应该进行同步修改。利用while 循环不断处理用户端传来的信息。如果用户端传来 "bye" 字符串，则进行退出操作。有如下伪代码，其实与任务 1 中类似：

```

while(1)
    rec_str_from_client(rec_str)
    if rec_str == "bye":
        exit
    get_str_for_answer(send_str, rec_str)
    send_str_to_client(send_str)

```

3、**处理请求：**实验要求按照客户给出的年份内容给出对应的生肖进行答复。在代码中写一函数，接受一个数字输入，返回一字符串，即为对应的生肖。可以使用 switch 方法进行实现。

4、**持续接受用户：**在一个用户输入 "bye" 之后，关闭服务端当前的 data_socket 接口。但是在全程我们应保持监听的 listen 接口保持开启。当 server 端被第一个用户占用时，若第二个用户接入，则会进入“排队”，第一个用户释放后服务器会立即接入第二个用户。若没有第二个用户，则继续进入监听状态，等待新用户接入。

```

void stop_now(){ // 解除当前用户
    printf("Stop now client\n");
    if(server_sock_data != -1 && close(server_sock_data) == -1)
        perror("server_sock_data close failed!\n");
    server_sock_data = -1;
    return ;
}

```

5、**Ctrl+C退出：**利用 signal 函数进行监听。在 Linux 系统中，Ctrl+C 会使进程收到 SIGINT 信号。若接收到该信号，则进行退出处理，关闭当前用户的 data_socket 接口与监听，然后结束程序。

```

void handle_exit(int sig){
    printf("\nCaught exit signal\n");
    if(server_sock_data != -1)
        if(close(server_sock_data) == -1)
            perror("server_sock_data close failed!\n");

    /* 关闭监听socket */
    if(server_sock_listen != -1 && close(server_sock_listen) == -1)
        perror("server_sock_listen close failed!\n");
    exit(0);
}

signal(SIGINT, handle_exit); // 识别 Ctrl C 退出

```

运行结果：

首先，第一个用户成功与服务端进行了对话。在第一个用户终止了对话后，第一个用户的程序成功退出。服务端继续进行等待，第二个用户再次发起请求后，服务端继续进行对话。最终服务端在收到 `Ctrl+C` 命令后成功退出。

```
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./server_a 12345
Socket created successfully.
Recv: 1234
Send: horse
Recv: abc
Send: Input error!
Recv: bye
Stop now client

Listening...
Recv: 1900
Send: rat
Recv: bye
Stop now client

Listening...
^C
Caught exit signal
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$

Connect Error!
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./client 127.0.0.1 12345
Socket created successfully.
ip addr: 127.0.0.1 port number: 12345
Input your messages: 1234
Send: 1234
Recv: horse
Input your messages: abc
Send: abc
Recv: Input error!
Input your messages: bye
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./client 127.0.0.1 12345
Socket created successfully.
ip addr: 127.0.0.1 port number: 12345
Input your messages: 1900
Send: 1900
Recv: rat
Input your messages: bye
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ S
```

此外，可以看到，该状态下服务端一次只能服务一个用户。在第一个用户占用了服务端之后，第二个用户得不到回复。

```
Listening...
^C
Caught exit signal
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./server_a 12345
Socket created successfully.
Recv: 1234
Send: horse

Recv: rat
Input your messages: bye
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./client 127.0.0.1 12345
Socket created successfully.
ip addr: 127.0.0.1 port number: 12345
Input your messages: 1234
Send: 1234
Recv: horse
Input your messages:

lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ cd Desktop/network
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./client 127.0.0.1 12345
Socket created successfully.
ip addr: 127.0.0.1 port number: 12345
Input your messages: 1234
Send: 1234
```

此外，在第一个用户退出链接之后，第二个用户则会立即得到服务端的回复，继续与服务端正常会话。

```
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./server_a 12345
Socket created successfully.
Recv: 1234
Send: horse
Recv: bye
Stop now client

Listening...
Recv: 1234
Send: horse
Recv: 666
Send: tiger

ip addr: 127.0.0.1 port number: 12345
Input your messages: 1900
Send: 1900
Recv: rat
Input your messages: bye
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./client 127.0.0.1 12345
Socket created successfully.
ip addr: 127.0.0.1 port number: 12345
Input your messages: 1234
Send: 1234
Recv: horse
Input your messages: bye
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$

lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ cd Desktop/network
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./client 127.0.0.1 12345
Socket created successfully.
ip addr: 127.0.0.1 port number: 12345
Input your messages: 1234
Send: 1234
Recv: horse
Input your messages: 666
Send: 666
Recv: tiger
Input your messages:
```

服务端backlog

在服务器端，在绑定好端口等信息后，主要还有如下步骤来建立 Socket 链接：

- 通过调用listen函数使Socket转入监听状态，等待来自客户端的连接请求
- 收到客户端发送的SYN报文后，TCP状态切换为SYNRECEIVED，并发送SYNACK报文
- 收到客户端发送的ACK报文后，TCP三次握手完成，状态切换为ESTABLISHED

在Linux系统中，使用两个队列syn queue, accept queue分别存储状态为SYN_RECV和ESTABLISHED的连接，backlog表示accept queue的大小。

调整 *backlog* 参数, 得到的 *netstat* 统计结果如下二图所示:

```
lzy@lzy-VMware-Virtual-Platform:~$ netstat -anp | grep 12345
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp        1      0 0.0.0.0:12345          0.0.0.0:*              LISTEN
42288/./server_a
tcp        0      0 127.0.0.1:36644        127.0.0.1:12345        ESTABLISHED
42338/./client
tcp        0      0 127.0.0.1:12345        127.0.0.1:56532        ESTABLISHED
42288/./server_a
tcp        0      1 127.0.0.1:50082        127.0.0.1:12345        SYN_SENT
42354/./client
tcp        0      1 127.0.0.1:36608        127.0.0.1:12345        SYN_SENT
42388/./client
tcp        0      0 127.0.0.1:12345        127.0.0.1:36644        ESTABLISHED
-
tcp        0      0 127.0.0.1:56532        127.0.0.1:12345        ESTABLISHED
42310/./client
lzy@lzy-VMware-Virtual-Platform:~$
```

(a) *backlog* == 0

```
lzy@lzy-VMware-Virtual-Platform:~$ netstat -anp | grep 12345
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp        2      0 0.0.0.0:12345          0.0.0.0:*              LISTEN
44515/./server_a
tcp        0      0 127.0.0.1:12345        127.0.0.1:54738        ESTABLISHED
-
tcp        0      0 127.0.0.1:54724        127.0.0.1:12345        ESTABLISHED
44525/./client
tcp        0      0 127.0.0.1:12345        127.0.0.1:54710        ESTABLISHED
44515/./server_a
tcp        0      1 127.0.0.1:46418        127.0.0.1:12345        SYN_SENT
44528/./client
tcp        0      0 127.0.0.1:54710        127.0.0.1:12345        ESTABLISHED
44516/./client
tcp        0      0 127.0.0.1:12345        127.0.0.1:54724        ESTABLISHED
-
tcp        0      0 127.0.0.1:54738        127.0.0.1:12345        ESTABLISHED
44527/./client
lzy@lzy-VMware-Virtual-Platform:~$
```

(b) *backlog* == 1

不难发现, 在设置 *backlog* 为 0 的时候, 有 2 对连接处于 ESTABLISHED 状态。在设置 *backlog* 为 1 的时候, 有 3 对连接处于 ESTABLISHED 状态。可以发现, 处于 ESTABLISHED 状态的连接数量增多, 设置的 *backlog* 参数增加了队列的容量。

端口字节顺序转换

在代码中注释掉端口字节顺序转换之后，会发现 client 端并不会正确接入到 server 端上。在经过一段时间后，client 端会提示连接错误，错误内容为连接超时。由此可见，对端口进行字节转换才能保证连接能够正常进行。

```
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./server_a 12345
Socket created successfully.
[]

lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./client 127.0.0.1 12345
Socket created successfully.
ip addr: 127.0.0.1 port number: 12345
Connect Error!: connection timed out
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ []
```

(a) 去掉字节顺序转换

```
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./server_b 123456
Socket created successfully.
Recv: 1234
Send: horse
Recv: bye
Stop now client
Listening...

lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ ./client 127.0.0.1 123456
Socket created successfully.
ip addr: 127.0.0.1 port number: 123456
Input your messages: 1234
Send: 1234
Recv: horse
Input your messages: bye
lzy@lzy-VMware-Virtual-Platform:~/Desktop/network$ []
```

(b) 正常情况

原因：

网络字节序

网络字节序是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。**网络字节序采用大端字节序。**

主机字节序

不同的机器主机字节序不相同，与CPU设计有关，数据的顺序是由cpu决定的，而与操作系统无关。我们把某个给定系统所用的字节序称为主机字节序（host byte order）。**对于 x86 CPU 都是小端字节序。**

恰好实验中我使用的电脑为 Intel x86 CPU，其字节序与网络字节序不同。因此，若不进行转换，则会导致端口号错误，从而无法成功建立连接。

Client 绑定端口

正常来讲，client 侧是不需要绑定端口的，因为系统会自动为用户侧分配端口。对于服务器来说，通过绑定到特定的端口，服务器能够在这个端口上等待并接收客户端发送的数据。

在实际测试中，如果在用户端也进行端口绑定，若输入和服务端绑定的相同端口，则会出现地址已经被使用的报错。若用户端随意选择其他可用端口，则会出现数据回环现象，即用户端发送“1234”，也会收到“1234”。从这两种情况不难看出，正常情况下用户端不应进行端口绑定，否则会造成错误。

在使用 netstat 进行观察时，可以发现 1234 端口和 1234 端口自己进入了 Established 状态，印证了上述数据回环现象。

```
if(bind(client_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) ==
-1){

    perror("Bind error!");
    return 0;
}
```

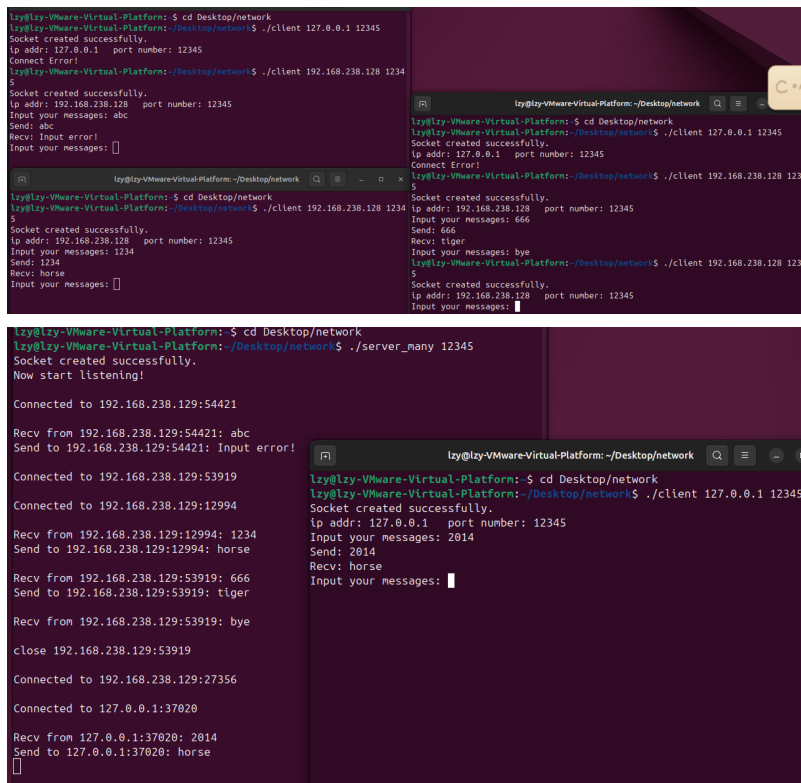


The screenshot shows three terminal windows. The first window runs ./server_a 12345 and shows 'Socket created successfully.'. The second window runs ./client 127.0.0.1 12345 and shows a successful connection and message exchange. The third window runs netstat -anp | grep 1234, showing two ESTABLISHED connections: one from 127.0.0.1:12345 to 127.0.0.1:12345, and another from 127.0.0.1:1234 to 127.0.0.1:1234, both associated with ./server_a.

任务3：年份生肖查询服务器(TCP并发)

运行结果

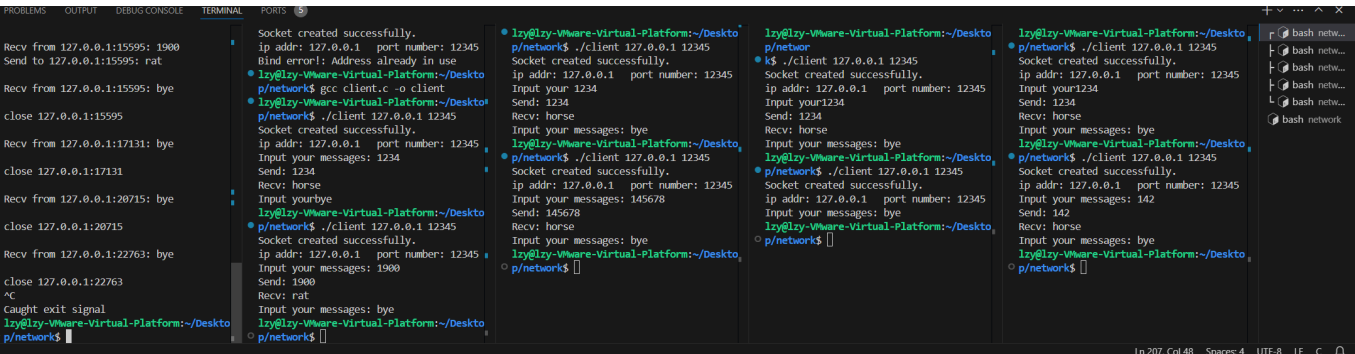
同时运行四个客户机，其中一个在同 server 的同虚拟机上，另外三个在复制的另一个虚拟机上。可以看到运行正常，服务端会正常返回信息，显示连接、断开等状况，以及用户端的 ip 地址、端口等信息。



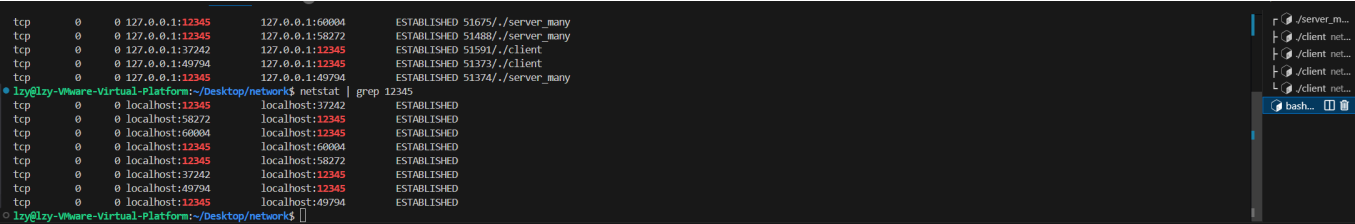
The screenshot shows a server terminal and two client terminals. The server terminal runs ./server_many 12345 and shows multiple connections from different IP addresses (192.168.238.129, 192.168.238.128, 127.0.0.1) and the corresponding messages sent and received. The client terminals show the process of creating a socket, connecting to the server, and sending/receiving messages.

父进程是否关闭 socket

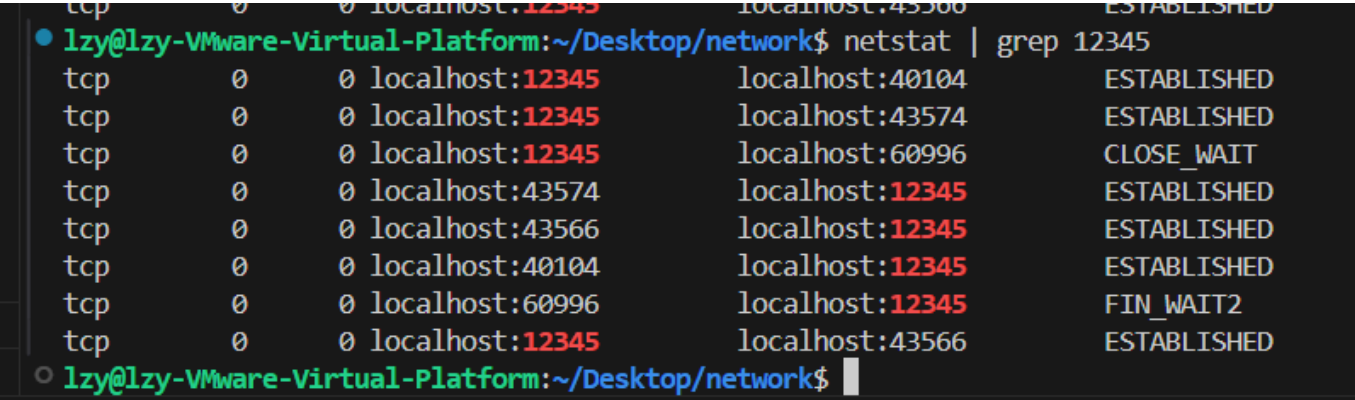
首先检查父进程不关闭是否会影响程序的运行。下图中，我启用了四个 client 端，且让父进程不关闭掉当前 socket。通过运行状态来看，似乎不会影响功能的实现。



然而，在使用 netstat 查看接口的状态时，发现二者有区别，如下二图所示。在主进程不关闭当前 socket 的情况下，如果用户端关闭掉一个会话，会发现保留有一个 "CLOSE_WAIT" 和 "FIN_WAIT2" 连接。



(a) 关闭



(b) 不关闭

理论分析：

在 Liunx 系统中，Fork 函数会直接复制当前进程的所有内容，并独立地创建一个全新的进程。两个进程是完全独立，互不影响的。在进程的复制过程中，会涉及到文件引用的复制。在子进程被创建之后，主进程对某一个文件的引用打开同样会被复制到子进程当中。

接下来涉及到 Liunx 系统下的文件系统。在 Liunx 系统中，每个文件会有 iNode 数据结构来记录当前文件的信息。每有一个新的进程引用了当前文件，则其 iNode 节点中的 iCount 字段会加 1

。相应地，如果每有一个进程释放 (close) 了当前文件，那么其 iNode 节点中的 iCount 字段会对应减 1。如果 iCount 为 0, 系统才会释放这个 iNode 以及其他资源。在 Socket 编程中，每一个 Socket 也相当于是一个文件。如果我们在父进程中不释放当前 socket，那么在子进程结束后，socket 的 iNode 中的 iCount 字段并不会变为 0, 因为系统认为父进程仍在用这一文件。最终，导致该 socket 一直得不到释放，从而造成系统资源的一大浪费。

综上所述，我们应当在父进程中及时关闭对应 socket。

任务4：使用Socket获取网页内容(Python)

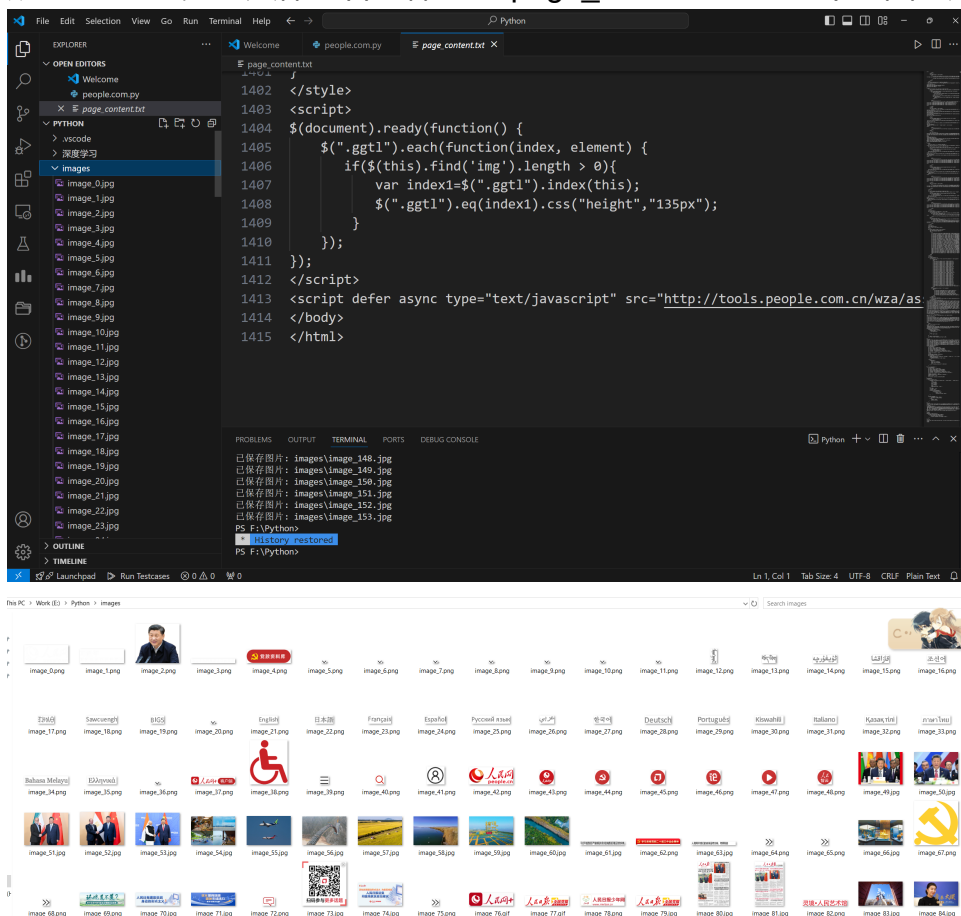
任务要求：

写程序，建立与 www.people.com.cn 的tcp连接，请求网页的内容并保存。

1. 将网页的内容以字符串形式保存在txt文件中。
2. (选做)将网页中的图片保存到本地文件夹。

运行结果：

所有 html 网页的具体内容被保存在 page_content.txt 文件中。图片则保存在文件夹 images 下。



代码解释：

首先是预处理部分。先创建一个 TCP/IP 的用户侧 socket，然后连接向指定的网站域名和端口（默认为 80）。

```
host = 'www.people.com.cn'
port = 80

# 创建一个TCP/IP socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((host, port))
```

接下来，向服务器发动一次 request 请求。

```
request = "GET / HTTP/1.1\r\nHost: www.people.com.cn\r\n\r\n"
client_socket.send(request.encode())
```

之后，利用缓冲区多次接受网页传来的内容，并不断保存到 response 字符串中。为了避免乱码问题，这里使用了 decode 方法将其转换为 "utf-8" 编码。最后使用 python 的文件操作将字符串保存到文本文档中。值得注意的是，相应内容包含头部以及数据本体。我们应当将其进行分离。

```
# 接收数据
response = b""
while True:
    data = client_socket.recv(10240) # 每次接收10240字节
    if not len(data):
        break
    response += data

client_socket.close()
response_text = response.decode('utf-8', errors='ignore')

header, body = response_text.split("\r\n\r\n", 1) # 分离响应内容

with open("page_content.txt", "w", encoding="utf-8") as f:
    f.write(body)
```

对于图片的处理，由于直接使用 socket 过于底层，较为复杂，我在这里使用了更加高级的第三方库 Request 来获取图片内容。

由于我们已经获取了网页的全部内容，因此我们可以直接使用正则表达式获取网页内容中的所有图片链接：

```
img_urls = re.findall(r'<img[^>]+src="([^">]+)"', body)
```

然后，利用 request 库来获取图片链接对应的内容：

```
        # 获取图片的响应内容
img_response = requests.get(img_url)
img_response.raise_for_status()
```

最后，处理一下图片后缀等问题即可：

```
# 从URL中提取文件后缀
parsed_url = urlparse(img_url)
img_ext = os.path.splitext(parsed_url.path)[1]

# 如果没有后缀，则默认使用.jpg
if not img_ext:
    img_ext = ".jpg"
img_name = os.path.join("images", f"image_{i}{img_ext}")

# 保存图片到文件
with open(img_name, 'wb') as img_file:
    img_file.write(img_response.content) # 保存图片内容
print(f"已保存图片: {img_name}")
```

实验总结

在这次实验中，我主要实践了在 Liunx 环境下进行的 *TCP/IP* 协议进行通信的 socket 编程以及使用 python 所编写的网页爬虫。

在第一项内容中，我通过对用户机、服务机两端的不断修改、改进，对 *TCP* 等协议的建立以及工作过程有了更加深刻、明确的认知，同时学会了使用 *netstat* 命令来查看接口的参数与状态。此外，这次实验也让我学习到了 Liunx 环境下 *fork* 函数的功能，并结合了其他课程中所学习到的 Unix 环境下文件系统 *iNode* 及相关知识，解释了在父进程中不释放 socket 所带来的系统资源浪费及其对应原因。此外，针对 *backlog* 参数、网络字节序转换、用户端绑定端口的实验，也让我对网络协议以及 Liunx 系统的运行细节有了更加深刻的认知。

在第二项实验内容中，我实现了利用 socket 以及 python 语言编写网络爬虫，成功爬取了网站的所有内容，并成功利用 python 的高级库获取了所有图片内容。这项实验让我对于编写网络爬虫这一工作变得更加的熟练。

附加解释

截图：为了编辑代码方便，实验后期我使用了 vscode 的 ssh 连接功能连接到了笔记本上的虚拟机，因而部分截图的命令行背景不一样。与在虚拟机图形界面运行环境完全相同。

完整代码：放在了另一文件夹下。其中，每个代码对应不同的任务。

- example.c: 这两份代码对应的是实验原本提供的样例代码。
- client.c: 修改后的客户端
- server.c: 对应任务一
- server_animal.c: 对应任务二
- server_many.c: 对应任务三
- people.com.py: 对应任务四