

网络安全技术

实 验 报 告

学 院 网络空间安全学院
年 级 2021
学 号 2112060
姓 名 孙 璐

目录

| | |
|------------------------|----|
| 一、 实验目的 | 1 |
| 二、 实验内容 | 1 |
| 三、 实验原理 | 1 |
| 四、 实验步骤 | 7 |
| 五、 实验结果 | 19 |
| 六、 实验遇到的问题及其解决方法 | 20 |
| 七、 实验结论 | 21 |

一、实验目的

本次实验目的如下：

- ① 理解 DES 加解密原理。
- ② 理解 TCP 协议的工作原理。
- ③ 掌握 linux 下基于 socket 的编程方法。

二、实验内容

利用 socket 编写一个 TCP 聊天程序，通信内容经过 DES 加密与解密。

- ① 能够在了解 DES 算法原理的基础上，编程实现对字符串的 DES 加密解密操作。
- ② 能够在了解 TCP 和 Linux 平台下的 Socket 运行原理的基础上，编程实现简单的 TCP 通信，为简化编程细节，不要求实现一对多通讯。
- ③ 将上述两部分结合到一起，编程实现通信内容事先通过 DES 加密的 TCP 聊天程序，要求双方事先互通密钥，在发送方通过该密钥加密，然后由接收方解密，保证在网络上传输的信息的保密性。

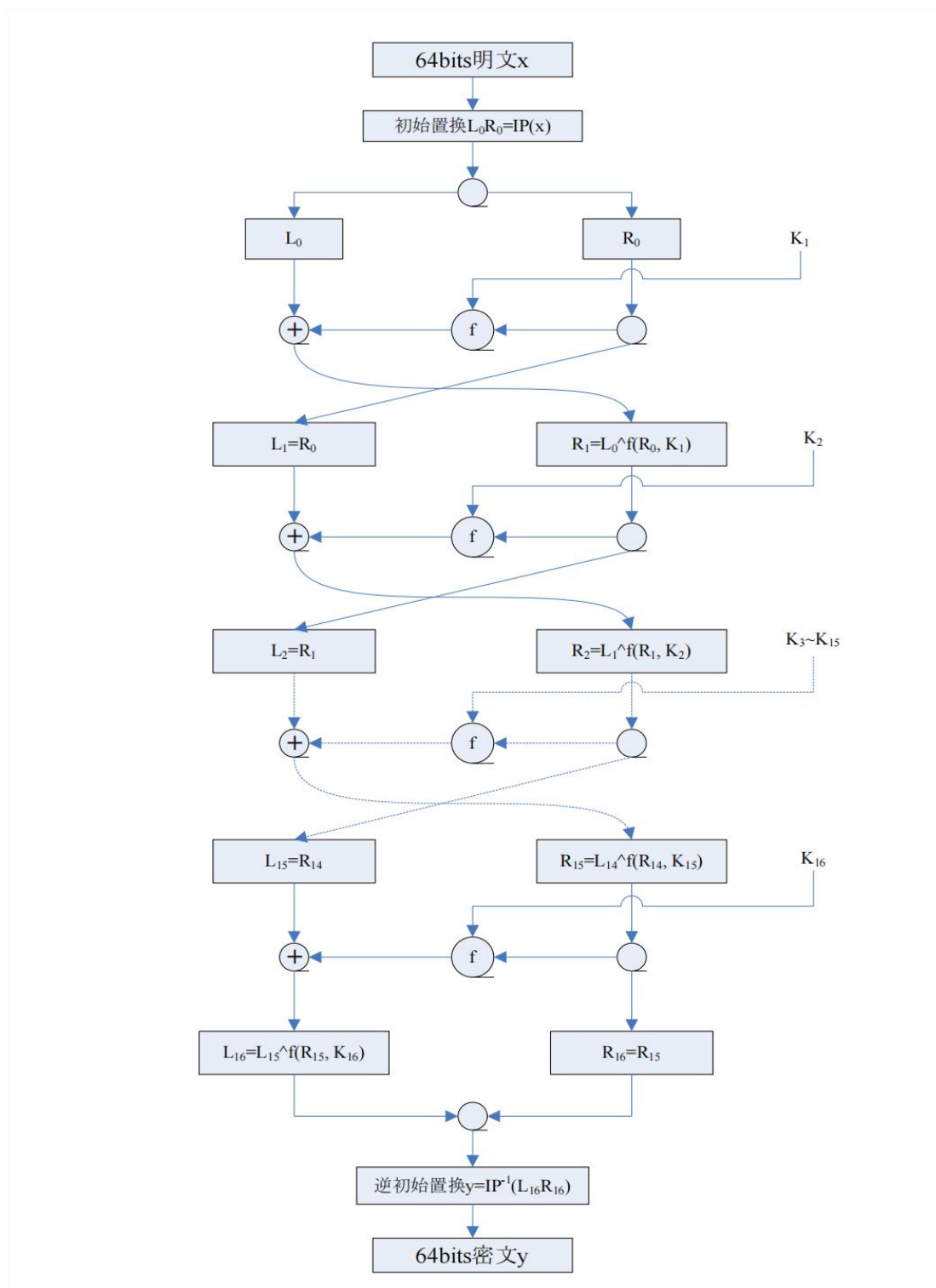
三、实验原理

1. DES

DES (Data Encryption Standard) 算法是一种用 56 位有效密钥来加密 64 位数据的对称分组加密算法，该算法流程清晰，已经得到了广泛的应用，算是应用密码学中较为基础的加密算法。

DES 使用 56 位的密钥和 64 位的明文块进行加密。DES 算法的分组大小是 64 位，如果需要加密的明文长度不足 64 位，需要进行填充；如果明文长度超过 64 位，则需要使用分组模式进行分组加密。

当输入了一条 64 位的数据之后，DES 将通过以下步骤进行加密。



(1) 初始置换

通过初始置换（IP 置换）将 64 位明文块进行置换和重新排列生成新的 64 位数据块。DES 中通过 IP 置换表完成初始置换 IP 操作，这个置换规则是固定的。观察 IP 置换表，发现相邻两列元素位置相差 8 位，前 32 位均为原奇数位号码，

后 32 个均为原偶数位号码。IP 置换目的是将输入的 64 位数据块按位重新组合，并把输出分为 L0、R0 两部分，每部分各长 32 位。

(2) 加密轮次

右半部分 R0 会作为下一轮次的左半部分 L1 的输入。其次，R0 会补位到 48 位和本轮次生成的 48 位 K0 输入到 F 函数中去。F 函数的输出结果为 32 位，结果 F(R0, K0) 会和 L0 进行异或运算作为下一轮次右半部分 R1 的输入。以此类推，需要重复 16 轮运算。

$$\begin{aligned} R_i &= F_i(R_{i-1}) + L_{i-1} \\ L_i &= R_{i-1} \end{aligned}$$

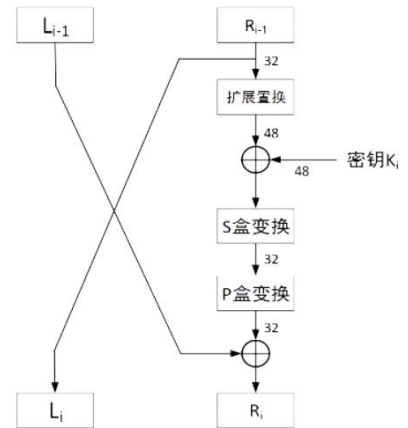
(3) F 函数

a) 选择扩展运算 (E 置换)

将 32 位的 R0 右半部分进行扩展，得到一个 48 位的数据块。这里的扩展表也是固定的。置换结果按行输出的结果即为密钥加运算 48 bit 的输入。

b) 密钥加运算

将 E 置换输出的 48 bit 作为输入，与 48 bit 子密钥进行异或运算。



c) 选择压缩运算 (S 盒)

S 盒替换是一种在密码学中广泛使用的加密技术。它是将明文中的一组比特映射到密文中的一组比特的过程，用于增强密码的安全性。

DES 中 S 盒运算用于将上一轮异或运算的 48 位结果映射到 32 位输出中。DES S 盒变换一共有 8 个 4 行 16 列的 S 盒表，记为 S1-S8，每一次 S 盒计算都有自己的变换表。

每个 S 表输入为 6bit，输出 4bit。在查表前，需要将密钥加运算得到的 48bit 输出分为 8 组，每组 6 bit，分别进入 8 个 S 盒表参与运算作为输入，得到的 8 个 4bit 的输出，将其串在一起的 32bit 输出作为置换运算 (P 盒) 的输入。

S 盒运算如下，S 盒输入的 6bit $b_1b_2b_3b_4b_5b_6$ ，取第一位 b_1 和最后一位 b_6 组成二

进制数 $r=b_1b_6$ ，其余位 $b_2b_3b_4b_5$ 组成二进制数 $c=b_2b_3b_4b_5$ 。查找对应的 S 盒表，找到第 $r+1$ 行 $c+1$ 列的数，转为二进制后即为该 S 盒运算要输出的结果。

d) 置换运算 (P 盒)

P 盒替换将 S 盒替换的 32 位输出作为输入，经过固定的替换表进行替换后即最后 F 轮函数的结果。该结果 $F(R_i, K_i)$ 与 L_i 进行异或运算得到下一轮的右半部分 R_{i+1} ，进行加密操作前原始的 R_i 将作为下一轮的 L_{i+1} 。

e) 逆初始置换 IP^{-1}

在 16 轮迭代后，将左右两端合并为 64bit，进行逆初始置换得到输出的 64bit 密文。

DES 算法采用了每轮子密钥生成的方式来增加密钥的复杂性和安全性。每轮子密钥都是由主密钥 (64 位) 通过密钥调度算法生成的。DES 算法的密钥调度算法可以将 64 位的主密钥分成 16 个子密钥，每个子密钥 48 位，用于每轮加密中与输入数据进行异或运算。

(4) PC-1 置换

PC-1 置换是对用户输入的种子密钥 K 进行的操作，只在第一轮子密钥的产生过程中使用，目的是删除原来 64 位的密钥 K 中的奇偶校验位 (每 8 位数据的最后一位)，并将剩余的 56 位打乱重排。奇偶校验位对密钥的安全性没有贡献，删除后能使 DES 运算加快。

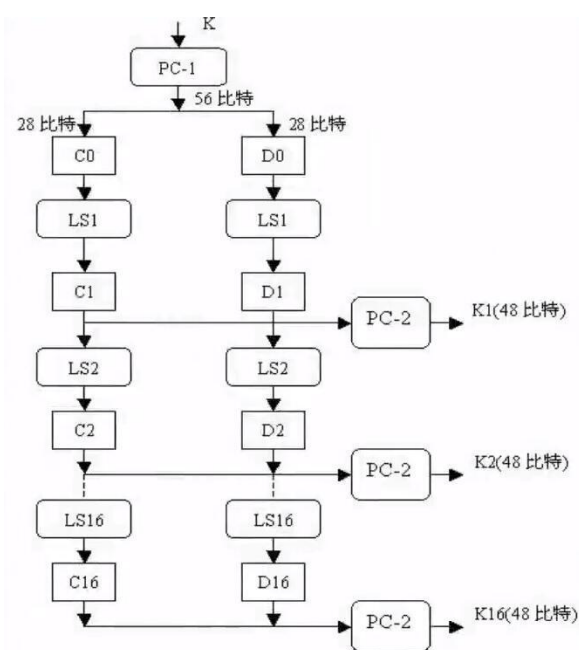
将 56 位的密钥分为左右两部分 28 bit C_0 、 D_0 ，通过 PC-1 的置换表进行循环左移操作。

(5) 循环左移

对 C_{i-1} 和 D_{i-1} 进行左移循环。对于 $i=1, 2, \dots, 16$ ，若 i 为 1, 2, 9 或 16，则循环左移一位，否则循环左移两位得到 C_i 和 D_i 。

(6) PC-2 置换

PC-2 置换的输入是由 PC-1 置换生成的 56 位的密钥，而它的输出是 48 位的子密钥。将输入的 56bit 中的第 9、18、22、25、35、38、43、54 位丢弃，按



着 PC-2 置换表进行置换。

在整个 DES 加密过程中会生成 16 个 48 位子密钥 K1-K16，分别用于 DES 算法中的 16 轮加密过程，从而保证每轮加密所使用的密钥都是不同的，增加了破解的难度。

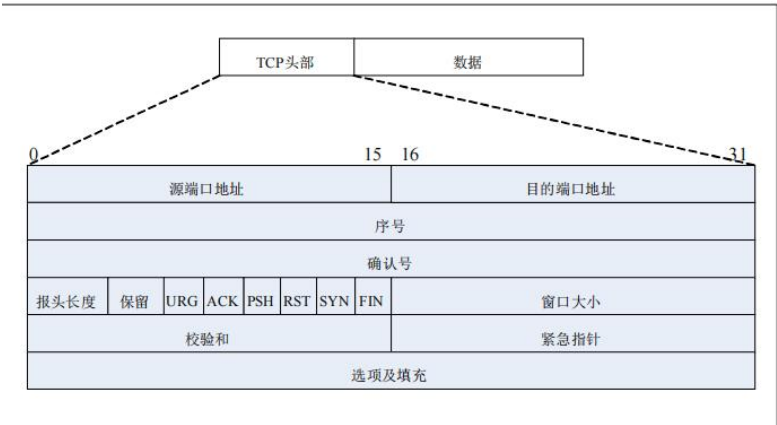
2. TCP

TCP（传输控制协议）是一种面向链接的、可靠的传输层协议。TCP 协议在网络层 IP 协议的基础上，向应用层用户进程提供可靠的、全双工的数据流传输。

TCP 协议每发送一个数据包将会收到一个确认信息。这种发送/应答模式可以使对方知道你是否收到了数据，从而保证了可靠性，当然，这也造成一些性能损失。为了改善系统效率不高的状况，TCP 引入了“捎带确认(piggybacking ACKs)”的方法。TCP 协议之所以是全双工的就是因为“捎带确认”信息，它允许双方同时发送数据，这是通过在当前的数据包中携带以前收到的数据的确认信息的方式实现的。从提高网络利用率的角度看，这比单纯发送一个通知对方“信息已收到”的数据包要好得多。同时，这种确认是批量确认的概念，即一次确认一个以上的数据包，表示“我收到了包括这个数据包在内的全部数据包”。

一个 TCP 段就是一个单个的 TCP 数据包。TCP 是一个数据流，因此，最关键的就是“连接”。最大报文段长度(MSS)是在连接的时候协商的，但是，它总是在不断地改变。默认的最大报文段长度是 536 字节，这是 576 字节(IP 协议保证的最小数据包长度)减去用于 IP 头的 20 个字节和用于 TCP 头的 20 个字节以后的长度。TCP 协议要设法避免在 IP 级别上的分段。

TCP 数据包的头 20 字节，各字段如下

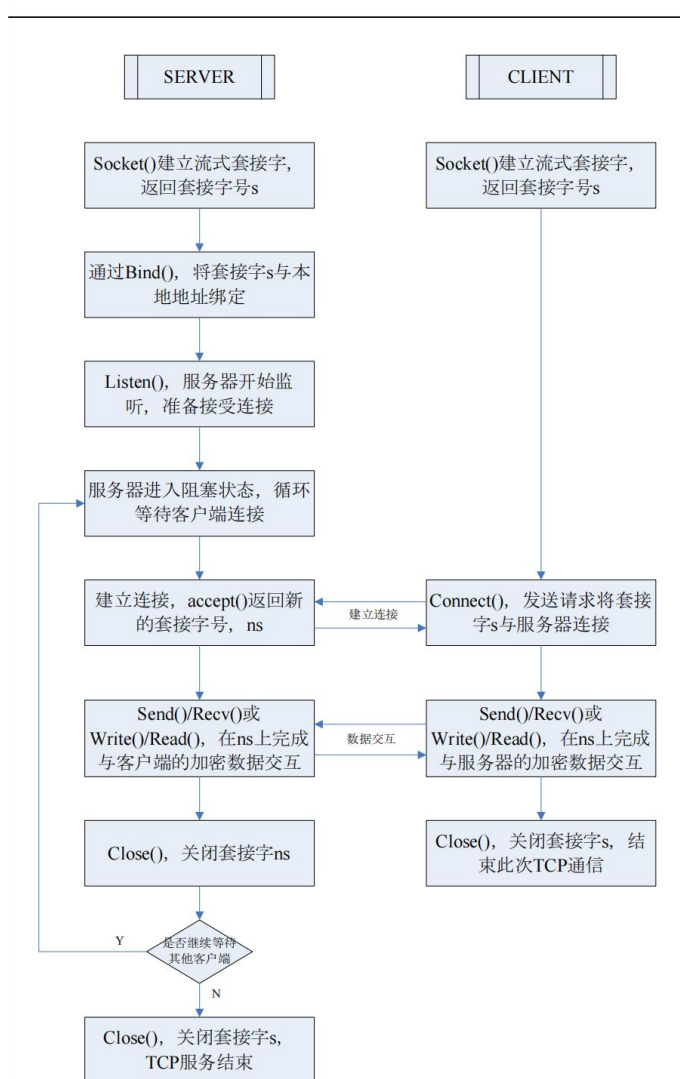


3. Socket 编程

Linux 系统是通过套接字（socket）来进行网络编程的。网络程序通过 socket 和其它几个函数的调用，会返回一个通讯的文件描述符，程序员可以将这个描述符看成普通的文件的描述符来操作。

套接字（Socket）是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口，它把复杂的 TCP/IP 协议族隐藏在 Socket 接口的背后，通过 Socket 函数调用去传输数据以符合指定的协议。套接字存在于通信区域中，一个套接字只能与同一区域内的套接字交换数据。TCP/IP 提供了三种类型的套接字，本次实验中使用的是流式套接字。

本次实验中需要用到 socket 函数以创建通信套接字，并返回该套接字的文件描述符；bind 函数于将套接字与指定端口相连；listen 函数以实现服务器等待客户端请求的功能；accept 函数用于处于监听状态的服务器，在获得客户机连接请求后，会将其放置在等待队列中，当系统空闲时，服务器用该函数接受客户机连接请求；connect 函数用于向客户端 socket 返回文件描述符；write 和 recv 函数用于服务器和客户端建立连接后，将 buf 中的 n bytes 字节的内容写入文件描述符；read 和 recv 函数从文件描述符 fd 中读取内容；close 函数用于关闭套接字。



四、 实验步骤

1. 建立 TCP 连接

```
//创建一个Socket来监听客户端的连接请求。
int serverPort = 5500;
int serverSocket = createServerSocket(serverPort);

if (serverSocket < 0)
{
    cerr << "Server: Failed to create a server socket. Error: " << strerror(errno) << endl;
    // cleanupWinSock();
    return 1;
}
else
{
    cerr << "Server: Create a server socket. " << endl;
}

//等待客户端的连接
sockaddr_in clientAddr;
socklen_t clientAddrLen = sizeof(clientAddr);

client_Socket = acceptClient(serverSocket, &clientAddr, &clientAddrLen);
```

(1) Server 端

a) 在 createServerSocket 函数中，建立流式套接字

```
// 创建服务器套接字
int createServerSocket(int port)
{
    cout << "commoncode: Creating server socket..." << endl;
    // 创建一个套接字，使用IPv4地址族(AF_INET)，流式套接字类型(SOCK_STREAM)，和TCP协议(IPPROTO_TCP)
    int serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (serverSocket < 0)
    {
        cerr << "commoncode: Create socket failed. Error: " << strerror(errno) << endl;
        return -1; // 返回-1表示创建套接字失败
    }
}
```

b) 配置服务器地址结构，进行 Bind 绑定和 Listen 监听，服务器进入阻塞状态，准备接受 client 端的连接

```
//将套接字绑定到指定的端口和IP地址
if (bind(serverSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) < 0)
{
    cerr << "commoncode: Bind failed. Error: " << strerror(errno) << endl;
    closeSocket(serverSocket);
    return -2;
}
else
{
    cout << "commoncode: Bind." << endl;
}

// 开始监听连接请求，最多允许10个等待连接
if (listen(serverSocket, 10) < 0)
{
    cerr << "commoncode: Listen failed. Error: " << strerror(errno) << endl;
    closeSocket(serverSocket);
    return -3; // 返回-3表示监听失败
}
else
{
    cout << "commoncode: Listen." << endl;
}

return serverSocket; // 返回创建的服务器套接字
```

c) 与客户端建立连接 accept()

```

//等待客户端的连接
sockaddr_in clientAddr;
socklen_t clientAddrLen = sizeof(clientAddr);

client_Socket = acceptClient(serverSocket, &clientAddr, &clientAddrLen);

```

```

// 接受客户端连接
int acceptClient(int serverSocket, sockaddr_in* clientAddr, socklen_t* clientAddrLen)
{
    cout << "commoncode: Accepting client connection..." << endl;
    // 使用 accept 函数接受客户端的连接请求, 返回一个新的套接字用于与客户端通信
    int clientSocket = accept(serverSocket, (struct sockaddr*)clientAddr, clientAddrLen);
    // 检查是否接受连接请求失败
    if (clientSocket < 0)
    {
        cerr << "commoncode: Accept failed. Error: " << strerror(errno) << endl;
        return -1; // 返回-1表示接受连接失败
    }
    cout << "commoncode: Client connected." << endl;
    return clientSocket; // 返回与客户端通信的套接字
}

```

(2) Client 端

a) 在 createServerSocket 函数中, 建立流式套接字

```

// 创建Socket: 在客户端创建一个Socket来连接服务器。
int clientSocket = createClientSocket("127.0.0.1", 5500);
/*if ((clientSocket == -2) || (clientSocket == -3) || (clientSocket == -4))
{
    closeSocket(clientSocket); // 关闭客户端套接字
    // cleanupWinSock(); // 清理WinSock库
    return 0;
}*/

```

b) connect 发送请求将套接字与服务器连接

```

// 创建客户端套接字
int createClientSocket(const char* serverIP, int serverPort)
{
    int clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (clientSocket == -1)
    {
        cerr << "commoncode: Failed to create a client socket. Error: " << strerror(errno) << endl;
        // cleanupWinSock();
        return -2;
    }

    // 连接服务器
    sockaddr_in serverAddr();
    bzero(&serverAddr, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    inet_pton(AF_INET, serverIP, &serverAddr.sin_addr);
    serverAddr.sin_port = htons(serverPort);

    // 尝试连接到服务器
    if (connect(clientSocket, (sockaddr*)&serverAddr, sizeof(serverAddr)) < 0)
    {
        cerr << "commoncode: Failed to connect to the server. Error: " << strerror(errno) << endl;
        closeSocket(clientSocket);
        // cleanupWinSock();
        return -4;
    }
    else
    {
        cout << "commoncode: Connect to the server." << endl;
    }

    return clientSocket;
}

```

(3) Recv 和 send 消息分别一个线程, 分别使用 recv() 和 send() 函数进行数据交互, 互相独立。需要注意的是, 本次实验在信息的收发前需要进行信息的加密和解密, 使用的是 pthread_create 函数:

```
int pthread_create(pthread_t* restrict tidp, const pthread_attr_t* restrict attr, void* (*start_rtn)(void*), void *restrict arg);
```

输入参数: tidp: 事先创建好的 pthread_t 类型的参数。成功时 tidp 指向的内存单元被设置为新建线程的线程 ID。attr: 用于定制各种不同的线程属性。通常直接设为 NULL。start_rtn: 新建线程从此函数开始运行。arg: start_rtn 函数的参数。无参数时设为 NULL 即可。有参数时输入参数的地址。当多于一个参数时应当使用结构体传入。如果创建成功则

返回 0, 否则返回错误码。

```
pthread_t cli_recv, cli_send;
pthread_create(&cli_recv, NULL, Client_recv_thread, NULL);
pthread_create(&cli_send, NULL, Client_send_thread, NULL);
```

```
pthread_t sev_recv, sev_send;
pthread_create(&sev_recv, NULL, Server_recv_thread, NULL);
pthread_create(&sev_send, NULL, Server_send_thread, NULL);
```

等待线程退出则使用的 pthread_join 函数

```
int pthread_join(pthread_t thread, void **retval);
```

输入参数: pthread_t thread: 被连接线程的线程号; void **retval : 指向一个指向被连接线程的返回码的指针的指针。返回值为线程连接的状态, 0 是成功, 非 0 是失败

当调用 pthread_join() 时, 当前线程会处于阻塞状态, 直到被调用的线程结束后, 当前线程才会重新开始执行。当 pthread_join() 函数返回后, 被调用线程才算真正意义上的结束, 它的内存空间也会被释放 (如果被调用线程是非分离的)

```
pthread_join(sev_recv, NULL);
pthread_join(sev_send, NULL);
```

```
pthread_join(cli_recv, NULL);
pthread_join(cli_send, NULL);
```

a) Client 的发送线程

```
void* Client_send_thread(void*)
{
    char client_send_buffer[BUFFER_SIZE];

    while (!exit_flag)
    {
        // 发送消息给服务器
        memset(client_send_buffer, 0, BUFFER_SIZE);
        cout << "Please input the message that the client will transmit: " << endl;
        cin.getline(client_send_buffer, BUFFER_SIZE - 1);
        if (strcmp(client_send_buffer, "exit") == 0)
        {
            exit_flag = true;
            break;
        }
        EncryptMsg(client_send_buffer, bkey);

        cout << "Client_send_thread: ciphertext that the client will transmit: " << client_send_buffer << endl;

        if (send(clientSocket, (void*)client_send_buffer, BUFFER_SIZE, 0) < 0)
        {
            cout << "Error: failed to send message to server." << endl;
            break;
        }
    }
    pthread_exit(NULL);
}
```

默认设定 exit 字符串为退出消息接发的标志，当输入不是 exit 时，将消息使用 send() 函数发给服务器。如果输入是 exit，标志 exit_flag=true，将退出 while 循环，关闭连接

b) Client 的接收线程

```
void* Client_recv_thread(void*)
{
    char client_recv_buffer[BUFFER_SIZE];
    // char receive_decrypt_buffer[BUFFER_SIZE];

    while (!exit_flag)
    {
        // 接收服务器发送的消息
        memset(client_recv_buffer, 0, BUFFER_SIZE);
        // memset(receive_decrypt_buffer, 0, BUFFER_SIZE);

        int recv_len = recv(clientSocket, (void*)client_recv_buffer, BUFFER_SIZE, 0);
        cout << "Client_recv_thread: The ciphertext that received from server: " << client_recv_buffer << endl;
        if (recv_len < 0)
        {
            cout << "Failed to receive message from server." << endl;
            break;
        }
        if (recv_len == 0)
        {
            cout << "Server closed the connection." << endl;
            exit(0);
            break;
        }

        // char *receive_decrypt_buffer= DecryptMsg(receive_buffer, bkey);

        // 显示服务器发送的消息
        // cout << "Received message from server: plaintext: " << receive_decrypt_buffer << endl;

        // 显示服务器发送的消息
        DecryptMsg(client_recv_buffer, bkey);
        cout << "Client_recv_thread: plaintext that the client received from server:";
        cout << client_recv_buffer << endl;
    }
    pthread_exit(NULL);
}
```

默认设定 exit 字符串为退出消息接发的标志，使用 recv() 函数接收消息，并使用 DecryptMsg 进行消息解密。当接受到服务器的字符串是 exit，标志

exit_flag=true, 将退出 while 循环, 关闭连接。

c) Server 的发送线程

```
void* Server_send_thread(void*)
{
    char server_send_buffer[BUFFER_SIZE];

    while (!exit_flag)
    {
        // 发送消息给客户端
        memset(server_send_buffer, 0, BUFFER_SIZE);
        cout << "Please input the message that the server will transmit: " << endl;
        cin.getline(server_send_buffer, BUFFER_SIZE - 1);

        if (strcmp(server_send_buffer, "exit") == 0)
        {
            exit_flag = true;
            break;
        }

        EncryptMsg(server_send_buffer, bkey);

        cout << "Server_send_thread: ciphertext that the server will transmit: " << server_send_buffer << endl;

        if (send(client_Socket, (void*)server_send_buffer, BUFFER_SIZE, 0) < 0)
        {
            cout << "Error: failed to send message to client." << endl;
            break;
        }

        pthread_exit(NULL);
    }
}
```

默认设定 exit 字符串为退出消息接发的标志, 当输入不是 exit 时, 将消息使用 send() 函数发给客户端。如果输入是 exit, 标志 exit_flag=true, 将退出 while 循环, 关闭连接

d) Server 的接收线程

```
char server_recv_buffer[BUFFER_SIZE]{};

while (!exit_flag)
{
    // 接收客户端发送的消息
    memset(server_recv_buffer, 0, BUFFER_SIZE);
    int recv_len = recv(client_Socket, (void*)server_recv_buffer, BUFFER_SIZE, 0);

    cout << "Server_recv_thread: The ciphertext that received from client: " << server_recv_buffer << endl;

    if (recv_len < 0)
    {
        cout << "Failed to receive message from client." << endl;
        break;
    }

    if (recv_len == 0)
    {
        cout << "Client closed the connection." << endl;
        exit(0);
        break;
    }

    // char* recv_decrypt_buffer = DecryptMsg(server_recv_buffer, bkey);
    // 显示客户端发送的消息
    // cout << "Received message from client: plaintext: " << recv_decrypt_buffer << endl;

    // 显示客户端发送的消息
    DecryptMsg(server_recv_buffer, bkey);
    cout << "Server_recv_thread: plaintext that the server received from client: ";
    cout << server_recv_buffer << endl;

    pthread_exit(NULL);
}
```

默认设定 exit 字符串为退出消息接发的标志, 使用 recv() 函数接收消息, 并使用 DecryptMsg 进行消息解密。当接受到客户端的字符串是 exit, 标志 exit_flag=true, 将退出 while 循环, 关闭连接。

2. DES 算法的实现

因为 DES 中需要用到的表是固定不变的, 因此首先将需要使用到的置换表等

以 const 数组形式写在 DES.cpp 文件中。由于需要用到的表很多，下面只以 IP 置换表、逆初始置换表、选择扩展置换表为例进行展示。

```
16 //初始置换表(IP置换)
17 const static unsigned char IP_table[64] =
18 {
19     58, 50, 42, 34, 26, 18, 10, 2,
20     60, 52, 44, 36, 28, 20, 12, 4,
21     62, 54, 46, 38, 30, 22, 14, 6,
22     64, 56, 48, 40, 32, 24, 16, 8,
23     57, 49, 41, 33, 25, 17, 9, 1,
24     59, 51, 43, 35, 27, 19, 11, 3,
25     61, 53, 45, 37, 29, 21, 13, 5,
26     63, 55, 47, 39, 31, 23, 15, 7
27 };
28
29 //逆初始置换
30 const static unsigned char IP_reverse_table[64] =
31 {
32     40, 8, 48, 16, 56, 24, 64, 32,
33     39, 7, 47, 15, 55, 23, 63, 31,
34     38, 6, 46, 14, 54, 22, 62, 30,
35     37, 5, 45, 13, 53, 21, 61, 29,
36     36, 4, 44, 12, 52, 20, 60, 28,
37     35, 3, 43, 11, 51, 19, 59, 27,
38     34, 2, 42, 10, 50, 18, 58, 26,
39     33, 1, 41, 9, 49, 17, 57, 25
40 };
41
42 //选择扩展置换变换表(E盒)
43 static const unsigned char Extend_table[48] =
44 {
45     32, 1, 2, 3, 4, 5,
46     4, 5, 6, 7, 8, 9,
47     8, 9, 10, 11, 12, 13,
48     12, 13, 14, 15, 16, 17,
49     16, 17, 18, 19, 20, 21,
50     20, 21, 22, 23, 24, 25,
51     24, 25, 26, 27, 28, 29,
52     28, 29, 30, 31, 32, 1
53 };
```

(1) IP 置换

```
// 将 64 bit 的明文重新排列，而后分成左右两块，每块 32bit，用 left 和 right 表示。
// IP置换表中的数据指的是位置，例如IP_table[0]=58指将明文第58位放置第1位
// 观察IP置换表，发现相邻两列元素位置相差8位，前32位均为原奇数位号码，后32个均为原偶数位号码。
// 各列经过偶采样和奇采样置换后，再对其进行逆序排列，将阵中元素按行读出以便构成置换的输出。
// 右半部分会直接作为下一轮次的左半部分的输入

int IP(const Block& plain_block, HBlock& left_block, HBlock& right_block)
{
    // 第一个循环遍历right数组的每个元素，根据IP_table中的值从block中取出相应的元素赋值给right。
    for (int i = 0; i < right_block.size(); ++i)
    {
        right_block[i] = plain_block[IP_table[i] - 1]; // 置换后的right部分
    }
    for (int i = 0; i < left_block.size(); ++i)
    {
        left_block[i] = plain_block[IP_table[i + left_block.size()] - 1]; // 置换后的left部分
    }
    return 0;
}
```

通过初始置换（IP 置换）将 64 位明文块进行置换和重新排列生成新的 64 位数据块。DES 中通过 IP 置换表完成初始置换 IP 操作，这个置换规则是固定的。观察 IP 置换表，发现相邻两列元素位置相差 8 位，前 32 位均为原奇数位号码，后 32 个均为原偶数位号码。IP 置换目的是将输入的 64 位数据块按位重新组合，并把输出分为 L0、R0 两部分，每部分各长 32 位。

初始 IP 置换表的意思为把 64 位明文按照表中的规则替换，比如第一行，把

64 位明文的第 1 位换为其 58 位，第 2 位换为 50 位，第 3 位换位 42 位……使用 for 循环实现初始 IP 置换

为了方便后面循环中的函数的书写，此处 right_block 与 left_block 先交换了。

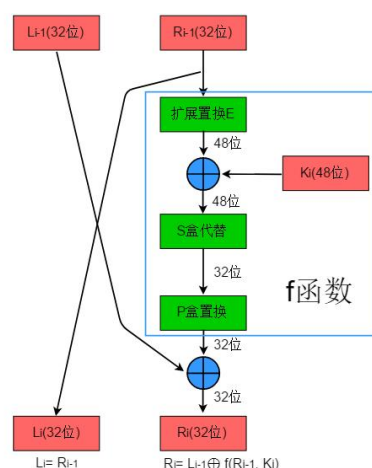
```
HBlock left_block, right_block; //左右部分
IP(block, left_block, right_block); //初始置换
```

(2) 加密轮次

右半部分 R0 会作为下一轮次的左半部分 L1 的输入。其次，R0 会补位到 48 位和本轮次生成的 48 位 K0 输入到 F 函数中去。F 函数的输出结果为 32 位，结果 F(R0, K0) 会和 L0 进行异或运算作为下一轮次右半部分 R1 的输入。以此类推，需要重复 16 轮运算。

```
case e: //加密
    for (char i = 0; i < 16; ++i)
    {
        Code key = getkey(i, bkey);
        Turn(left_block, right_block, key);
        if (i != 15)
        {
            exchange(left_block, right_block);
        }
    }
    break;
```

(3) 每一轮次中的 F 函数计算



a) 选择扩展运算（E 置换）

将 32 位的 R0 右半部分进行扩展，得到一个 48 位的数据块。这里的扩展表也是固定的。置换结果按行输出的结果即为密钥加运算 48 bit 的输入。使用 for 循环实现 E 置换。在 Extend_table 中获取扩展前 bit 所在的 right_block 位置，在 right_block 中找到这个 bit。

```
// 下面右半部分进行F函数运算
//将右半部分通过E盒扩展变换为48位
for (int i = 0; i < Extend_code.size(); ++i)
{
    Extend_code[i] = right_block[Extend_table[i] - 1]; //扩展置换
}
```

数据的扩展规则如下：中间为 32 位。两边为扩展位，扩展后为 48 位。

| | | | | | | | |
|----|--|----|----|----|----|--|----|
| 32 | | 01 | 02 | 03 | 04 | | 05 |
| 04 | | 05 | 06 | 07 | 08 | | 09 |
| 08 | | 09 | 10 | 11 | 12 | | 13 |
| 12 | | 13 | 14 | 15 | 16 | | 17 |
| 16 | | 17 | 18 | 19 | 20 | | 21 |
| 20 | | 21 | 22 | 23 | 24 | | 25 |
| 24 | | 25 | 26 | 27 | 28 | | 29 |
| 28 | | 29 | 30 | 31 | 32 | | 01 |

b) 密钥加运算

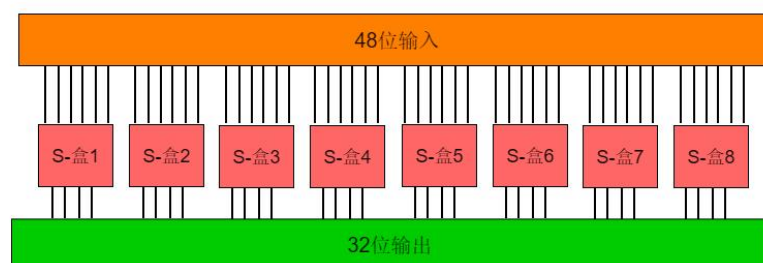
将 E 置换输出的 48 bit 作为输入，与 48 bit 子密钥进行异或运算。

```
// 扩展完成后进行密钥加运算，
// 将选择扩展运算输出的 48bit 作为输入，与 48bit 的子密钥进行异或运算，异或结果作为选择压缩运算（S 盒）的输入。
Extend_code ^= subkey; //与子密钥异或
```

c) 选择压缩运算（S 盒）

DES 中 S 盒运算用于将上一轮异或运算的 48 位结果映射到 32 位输出中。DES S 盒变换一共有 8 个 4 行 16 列的 S 盒表，记为 S1-S8，每一次 S 盒计算都有自己的变换表。

每个 S 表输入为 6bit，输出 4bit。在查表前，需要将密钥加运算得到的 48bit 输出分为 8 组，每组 6 bit，分别进入 8 个 S 盒表参与运算作为输入，得到的 8 个 4bit 的输出，将其串在一起的 32bit 输出作为置换运算（P 盒）的输入。




```

// 8个S盒，每个S盒有4行16列。
// 每个S盒输入为6bit，输出4bit。
// 在查表前，需要将密钥加密运算得到的48bit输出分为8组，每组6 bit，分别进入8个S盒参与运算作为输入。得到的8个4bit的输出，将其串在一起的32bit输出作为置换运算（P盒）的输入。

// S盒运算如下，S盒输入的6bit b1b2b3b4b5b6，取第一位b1和最后一位b6组成二进制数r = b1b6，其余位b2b3b4b5组成二进制数c = b2b3b4b5。
// 查找对应的S盒表，找到第r + 1行c + 1列的数，转为二进制后即为该S盒运算要输出的结果。

for (int i = 0; i < 8; ++i)
{
    row[0] = Extend_code[6 * i + 1]; //行标
    row[1] = Extend_code[6 * i + 5];

    col[0] = Extend_code[6 * i + 1]; //列标
    col[1] = Extend_code[6 * i + 2];
    col[2] = Extend_code[6 * i + 3];
    col[4] = Extend_code[6 * i + 4];

    // to_ulong()将row和col 两个bitset对象转换为unsigned long
    // s盒有4行16列，通过计算得到对应的转换值在s盒的哪个位置
    bitset<4> temp(S_Box[i][row.to_ulong() * 16 + col.to_ulong()]);

    // 将8组S盒计算得到的结果拼成一个32 bit 字符串
    // P盒替换需要按S盒替换的32位输出作为输入
    for (int j = 0; j < temp.size(); ++j)
    {
        Extend_code[4 * i + j] = temp[j];
    }
}

```

S 盒运算如下，S 盒输入的 6bit $b_1b_2b_3b_4b_5b_6$ ，取第一位 b_1 和最后一位 b_6 组成二进制数 $r=b_1b_6$ ，其余位 $b_2b_3b_4b_5$ 组成二进制数 $c=b_2b_3b_4b_5$ 。查找对应的 S 盒表，找到第 $r+1$ 行 $c+1$ 列的数，转为二进制后即为该 S 盒运算要输出的结果。

以 S1 盒为例，若 S1 的输入为 110111，第一位与最后一位构成 11，十进制值为 3，则对应第 3 行，中间 4 位为 1011 对应的十进制值为 11，则对应第 11 列。查找 S-盒 1 表的值为 14，则 S-盒 1 的输出为 1110。8 个 S 盒将输入的 48 位数据输出为 32 位数据。

| | | | | | | | | | | | | | | | |
|----|----|----|---|----|----|----|----|----|----|----|----|----|----|---|----|
| 14 | 4 | 13 | 1 | 2 | 15 | 11 | 8 | 3 | 10 | 6 | 12 | 5 | 9 | 0 | 7 |
| 0 | 15 | 7 | 4 | 14 | 2 | 13 | 1 | 10 | 6 | 12 | 11 | 9 | 5 | 3 | 8 |
| 4 | 1 | 14 | 8 | 13 | 6 | 2 | 11 | 15 | 12 | 9 | 7 | 3 | 10 | 5 | 0 |
| 15 | 12 | 8 | 2 | 4 | 9 | 1 | 7 | 5 | 11 | 3 | 14 | 10 | 0 | 6 | 13 |

根据上面的例子，在代码中将 48bit 分为 8 组，每一组的第一位和第六位为 $\text{Extend_code}[6*i]$ 与 $\text{Extend_code}[6*i+5]$ ，组成 2 bit 二进制数 row 作为行，中间四位为 $\text{Extend_code}[6*i+1]$ ， $\text{Extend_code}[6*i+2]$ ， $\text{Extend_code}[6*i+2]$ ， $\text{Extend_code}[6*i+3]$ ， $\text{Extend_code}[6*i+4]$ 组成 4 bit 二进制数 col 作为列，转为 10 进制，然后在 S 盒取对应位置的数。由于我在前面将 S 盒设成的是一维数组，在 S 盒对应的位置应该是 $\text{row}*16+\text{col}$ 。然后将 8 个 S 盒输出拼成 32bit 数据。

d) 置换运算（P 盒）

P 盒替换将 S 盒替换的 32 位输出作为输入，经过固定的替换表进行替换后即最后 F 轮函数的结果。

该结果 $F(R_i, K_i)$ 与 L_i 进行异或运算得到下一轮的右半部分 R_{i+1} ，进行加密操作前原始的 R_i 将作为下一轮的 L_{i+1} 。

```

// 将S盒计算得到的32位输出作为P盒的输入，经过替换表替换之后即位F轮函数的结果
for (int i = 0; i < P_code.size(); ++i)
{
    P_code[i] = Extend_code[P_table[i] - 1]; //P盒置换
}
// P盒计算得到的结果与上一轮左边的 32bit 进行异或作为下一轮的右半部分
left_block ^= P_code;

```

e) 逆初始置换 IP^{-1}

在16轮迭代后，将左右两端合并为64bit，进行逆初始置换得到输出的64bit密文。

末置换是初始置换的逆过程，DES 最后一轮后，左、右两半部分并未进行交换，而是两部分合并形成一个分组做为末置换的输入。

```

//将左右两部分数据进行逆初始置换并拼成一个数据块
int Reverse_IP(const HBlock& left_block, const HBlock& right_block, Block& block)
{
    for (int i = 0; i < block.size(); ++i)
    {
        if (IP_reverse_table[i] <= 32)
        {
            block[i] = right_block[IP_reverse_table[i] - 1]; //从right部分获取数据
        }
        else
        {
            block[i] = left_block[IP_reverse_table[i] - 32 - 1]; //从left部分获取数据
        }
    }
    return 0;
}

```

DES 算法采用了每轮子密钥生成的方式来增加密钥的复杂性和安全性。每轮子密钥都是由主密钥（64 位）通过密钥调度算法生成的。DES 算法的密钥调度算法可以将 64 位的主密钥分成 16 个子密钥，每个子密钥 48 位，用于每轮加密中与输入数据进行异或运算。

（4） 密钥生成

a) PC-1 置换

PC-1 置换是对用户输入的种子密钥 key 进行的操作，只在第一轮子密钥的产生过程中使用，目的是删除原来 64 位的密钥 key 中的奇偶校验位（每 8 位数据的最后一位，即 8, 16, 24, 32, 48, 56），并将剩余的 56 位打乱重排。奇偶校验位对密钥的安全性没有贡献，删除后能使 DES 运算加快。

在 PC1_table 中获取种子密钥 bkey 第 i bit 在 PC-1 置换后的位置，然后存在 subkey_pc1 的对应位置上。

```

//PC-1置换，得到56位密钥
for (int i = 0; i < subkey_pc1.size(); ++i)
{
    subkey_pc1[i] = bkey[PC1_table[i] - 1];
}

```

将 56 位的密钥分为左右两部分 28 bit C_0 、 D_0 ，通过 PC-1 的置换表进行循环左移操作。

b) 循环左移

对 C_{i-1} 和 D_{i-1} 进行左移循环。对于 $i=1, 2, \dots, 16$ ，若 i 为 1, 2, 9 或 16，则循环左移一位，否则循环左移两位得到 C_i 和 D_i 。

```
// 输出的56bit将被分成两组，每组28bit，分别进入C寄存器（pcl_table的上半部分）和D寄存器（pcl_table的下半部分），准备进行循环左移
for (int i = 0; i <= N; ++i)
{
    //第N轮密钥产生需要循环左移的位数为bit_leftshift的第N个元素的值

    int shift = bit_leftshift[N]; // 获取当前轮次的左移位数

    // 对两个28位部分分别进行循环左移
    // 0为左半部分，1为右半部分
    for (int part = 0; part < 2; ++part)
    {
        unsigned char temp[28]; // 临时存储左移后的结果
        for (int i = 0; i < 28; ++i)
        {
            int new_index = (i + shift) % 28; // 计算左移后的新位置
            temp[new_index] = subkey_pcl[part * 28 + i]; // 执行左移
        }
        // 将临时数组中的结果复制回原数组
        for (int i = 0; i < 28; ++i)
        {
            subkey_pcl[part * 28 + i] = temp[i];
        }
    }
}
```

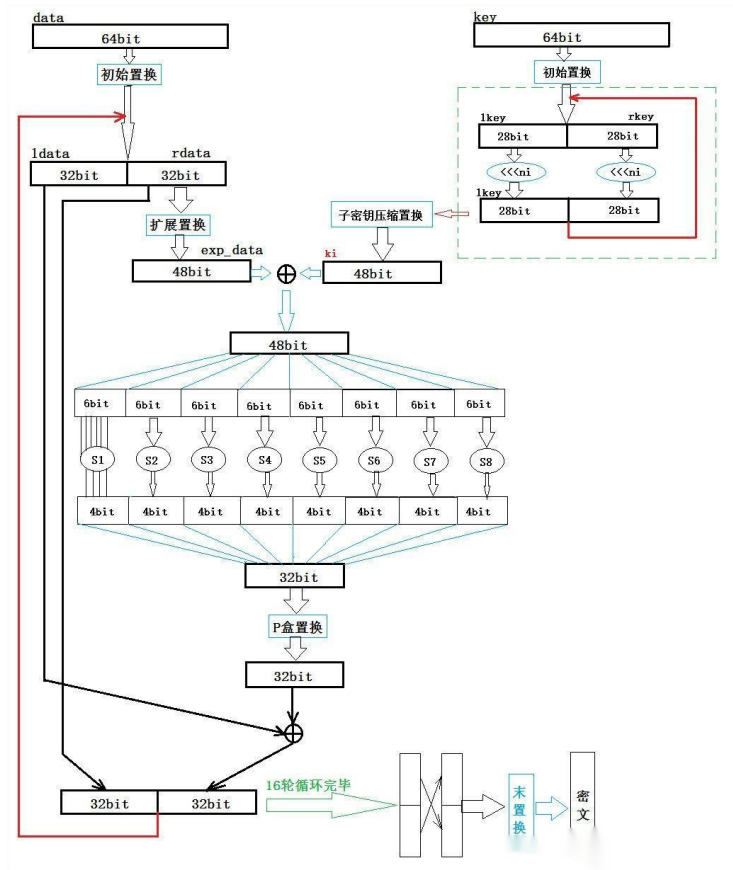
c) PC-2 置换

PC-2 置换的输入是由 PC-1 置换生成的 56 位的密钥，而它的输出是 48 位的子密钥。将输入的 56bit 中的第 9、18、22、25、35、38、43、54 位丢弃，按着 PC-2 置换表进行置换。

```
// PC-2置换
// 将56 bit中的第9、18、22、25、38、43、54位删除，其余位置按照PC2_table置换位置，将56bit密钥置换成48bit
for (int i = 0; i < subkey_pc2.size(); ++i)
{
    subkey_pc2[i] = subkey_pcl[PC2_table[i] - 1];
}
return subkey_pc2;
```

(5) 加解密过程

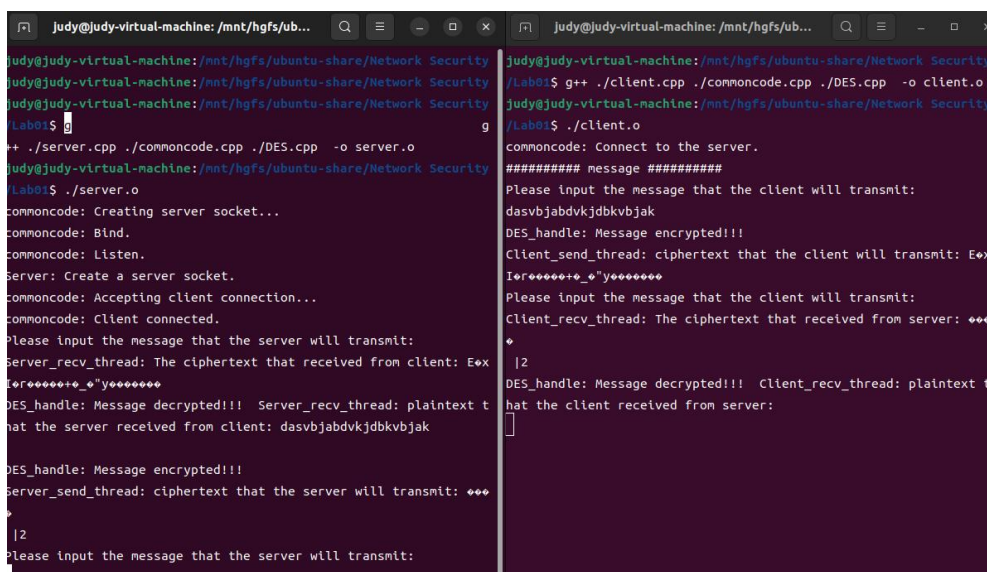
加密过程就是 (1) - (4) 的过程。解密过程与加密过程类似，唯一不同的是将 16 轮子密钥顺序反过来，即在加密过程中所用的加密密钥 k_1, \dots, k_{16} 在解密过程中需要转换成 k_{16}, \dots, k_1 。



```
//加解密运算
int des(Block& block, Block& bkey, const Method method)
{
    //block为数据块, bkey为64位密钥
    HBlock left_block, right_block; //左右部分
    IP(block, left_block, right_block); //初始置换

    switch (method)
    {
        case e: //加密
            for (char i = 0; i < 16; ++i)
            {
                Code key = getkey(i, bkey);
                Turn(left_block, right_block, key);
                if (i != 15)
                {
                    exchange(left_block, right_block);
                }
            }
            break;
        case d: //解密
            for (char i = 15; i >= 0; --i)
            {
                Code key = getkey(i, bkey);
                Turn(left_block, right_block, key);
                if (i != 0)
                {
                    exchange(left_block, right_block);
                }
            }
            break;
        default:
            break;
    }

    Reverse_IP(left_block, right_block, block); //末置换
    return 0;
}
```

```
judy@judy-virtual-machine: /mnt/hgfs/ubuntu-share/Network Security
judy@judy-virtual-machine: /mnt/hgfs/ubuntu-share/Network Security
judy@judy-virtual-machine: /mnt/hgfs/ubuntu-share/Network Security
/ Lab01$ g++ ./server.cpp ./commoncode.cpp ./DES.cpp -o server.o
judy@judy-virtual-machine: /mnt/hgfs/ubuntu-share/Network Security
/ Lab01$ ./server.o
commoncode: Creating server socket...
commoncode: Bind.
commoncode: Listen.
Server: Create a server socket.
commoncode: Accepting client connection...
commoncode: Client connected.
Please input the message that the server will transmit:
Server_rcv_thread: The ciphertext that received from client: E0x
I0r00000+0_0"Y00000000
DES_handle: Message decrypted!!! Server_rcv_thread: plaintext t
hat the server received from client: dasvbjabdkvjdskvbjak

DES_handle: Message encrypted!!!
Server_send_thread: ciphertext that the server will transmit: 000
02
Please input the message that the server will transmit:

judy@judy-virtual-machine: /mnt/hgfs/ubuntu-share/Network Security
/ Lab01$ g++ ./client.cpp ./commoncode.cpp ./DES.cpp -o client.o
judy@judy-virtual-machine: /mnt/hgfs/ubuntu-share/Network Security
/ Lab01$ ./client.o
commoncode: Connect to the server.
##### message #####
Please input the message that the client will transmit:
dasvbjabdkvjdskvbjak
DES_handle: Message encrypted!!!
Client_send_thread: ciphertext that the client will transmit: E0x
I0r00000+0_0"Y00000000
Please input the message that the client will transmit:
Client_rcv_thread: The ciphertext that received from server: 000
02
DES_handle: Message decrypted!!! Client_rcv_thread: plaintext t
hat the client received from server:
02
```

在任意一端输入待传递的消息后（截图中为 client 端输入待传输信息），通过 EncryptMsg 函数加密消息后，输出了加密后的字符串，可知已成功加密，然后通过客户端的发送线程发送到服务器端。

服务器端通过接收线程接收到了加密后的字符串，输出了接收到的加密字符串，与客户端加密的字符串一致，说明客户端与服务器端的发送是正常的。通过 DecryptMsg 进行字符串的解密，输出了解密后的字符串，可以看到与客户端发送的加密前的明文是一致的，说明 DES 加解密都正确实现。

六、 实验遇到的问题及其解决方法

本次实验中遇到了一些问题，但还是找到了解决办法。

（1） Linux 下 C++ Socket 编程与 Windows 下还是有一定不同的，两个环境下使用的库有的是不兼容的，有的函数在 Windows C++下能用但 Linux C++是不能使用的。比如创建线程的函数，Windows 中使用的是 CreateThread，而在 Linux 环境下使用的是 pthread_create。能使用的指针类型也不同，比如 Linux 下要用 pthread_t 类型，而 Windows 下可以使用 lparam 等。还有很多这样的例子，通过报错提示和网络查询，一步一步修改解决了类似的问题，进一步熟悉了 Linux 下 C++编程。

（2） Linux 下 C++编程环境搭建也遇到了一些问题。因为先前对 Linux 下 C++编程没有怎么接触，本学期本门课程是第一次。在 VS2022 中配置远程的 Linux 开发环境遇到了一些问题，通过搜索网络教程解决了环境配置的问题。

（3） 在调试 bug 的时候，部分函数出现异常报错，同样是通过网络教程，

一个一个解决了。

七、 实验结论

本次实验实现了 DES 加密的 TCP 聊天室的搭建，完成了实验要求。初次接触 Linux 下的 C++编程，初步了解与 Windows 下 C++编程的不同之处。通过配置环境遇到的问题及不断修复 bug，初步熟悉了 Linux 下的 C++编程。上学期在计算机网络课程中初次接触了 Socket 编程，在完成本次实验中进一步深入的理解了 Socket 编程。密码学 DES 加密算法的实现不仅深化了对各种加密算法的认识，而且与 Socket 编程的结合更加强了对密码学实用性的认识。