Socket 应用编程实验

学号： 2021K8009929010

姓名： 贾城昊

一、 实验题目：Socket 应用编程实验

二、 实验任务

使用 C 语言实现最简单的 HTTP 服务器，要求该服务器同时支持 HTTP（80 端口）和 HTTPS（443 端口），并使用两个线程分别监听各自端口。该服务器只需支持 GET 方法，解析请求报文，返回相应应答及内容。需要支持的状态码如下图所示：

需支持的状态码	场景
200 OK	对于443端口接收的请求，如果程序所在文件夹存在所请求的文件，返回该状态码，以及所请求的文件
301 Moved Permanently	对于80端口接收的请求，返回该状态码，在应答中使用Location字段表达相应的https URL
206 Partial Content	对于443端口接收的请求，如果所请求的为部分内容（请求中有Range字段），返回该状态码，以及相应的部分内容
404 Not Found	对于443端口接收的请求，如果程序所在文件夹没有所请求的文件，返回该状态码

三、 实验流程

1. 根据上述要求，实现 HTTP 服务器程序

2. 执行 sudo python topo.py 命令，生成包括两个端节点的网络拓扑

3. 在主机 h1 上运行 HTTP 服务器程序，同时监听 80 和 443 端口

4. h1 # ./http-server

5. 在主机 h2 上运行测试程序，验证程序正确性

   h2 # python3 test/test.py

   如果没有出现 AssertionError 或其他错误，则说明程序实现正确

6. 最后将代码等文件提交到 OJ 网站进行验证

四、 实验结果与分析

（一）服务器设计思路

首先，本人第一步是创建两个线程，分别用于支持 HTTP（80 端口）和 HTTPS（443 端口），这一部分在 main( )函数中完成，代码如下：

```c
int main()
{
    pthread_t thread1, thread2;

    if (pthread_create(&thread1, NULL, HTTP_SERVER, NULL) != 0)
    {
        perror("Thread creation failed");
        return -1;
    }

    if (pthread_create(&thread2, NULL, HTTPS_SERVER, NULL) != 0)
    {
        perror("Thread creation failed");
        return -1;
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

其中 HTTP_SERVER( )用于支持 HTTP，HTTPS_SERVER( )用于支持 HTTPS，下面是对这两个函数的具体介绍

1. HTTP_SERVER

HTTP 服务器首先是建立 socket 文件描述符,然后绑定监听地址，最后对 80 端口进行监听。然后是核心部分,循环不断接收连接请求,在每收到一个服务器的连接后,创建一个新的线程用于接收并解析其 HTTP 请求,然后根据请求内容进行应答。该部分具体代码如下所示:

```
while (1)
{
    struct sockaddr_in c_addr;
    socklen_t addr_len;

    int request = accept(sockfd, (struct sockaddr *)&c_addr, &addr_len);
    if (request < 0)
    {
        perror("Accept failed");
        exit(1);
    }

    pthread_t new_thread;

    if ((pthread_create(&new_thread, NULL, (void *)handle_http_request, (void *)&request)) != 0)
    {
        perror("Create handle_http_request thread failed");
        exit(1);
    }
}
```

其中与一个客户端建立连接后,使用 pthread_create( )创建一个线程调用 handle_http_request( )处理该线程的请求。而主线程继续执行循环,等待下一个连接的客户端。由此实现服务器的多线程多路并发处理。

http 请求的处理函数的功能描述如下:

首先是接受客户端的 request 报文，并检查是否为 GET 请求:

```
request_len = recv(request, recv_buff, 2000, 0);
if (request_len < 0)
{
    fprintf(stderr, "recv failed\n");
    exit(1);
}

char *req_get = strstr(recv_buff, "GET");
if (req_get)
{
```

然后根据 request 得到相应的 https URL（这里考虑了是相对的 URL 还是绝对的 URL），最后返回 301 Moved Permanently，并在应答中使用 Location 字段表达相应的 https URL，对应代码如下：

```
memset(send_buff, 0, 6000);
strcat(send_buff, http_version);
strcat(send_buff, " 301 Moved Permanently\r\nLocation: ");
strcat(send_buff, "https://");
```

上图用于返回 301 Moved Permanently 以及 Location 字段前缀

```
int i;
for (i = 0; (*iterator) != ' '; iterator++, i++)
{
    temp_url[i] = *iterator;
}
temp_url[i] = '\0';
iterator++;
```

```
if (relative_url)
{
    iterator = strstr(recv_buff, "Host:");
    if(!iterator){
        perror("Not found Host");
        exit(1);
    }
    iterator += 6;

    for (int i = 0; (*iterator) != '\r'; iterator++, i++)
    {
        host[i] = *iterator;
    }
    host[i] = '\0';
}
```

```
if (relative_url)
{
    strcat(send_buff, host);
    strcat(send_buff, temp_url);
}
else
{
    strcat(send_buff, &temp_url[7]);
}
strcat(send_buff, "\r\n\r\n\r\n\r\n");
```

上面三张图是用于从 request 中获取 URL，若为相对路径（即 GET 后为 / 开头），则获取 Host 字段后的信息，然后与相对路径的 URL 进行拼接，若为绝对路径，则直接写入响应信息里即可（前面的 https://字段之前统一写了，所以代码中绝对路径从第 8 个字符开始写入）。

```
if ((send(request, send_buff, strlen(send_buff), 0)) < 0)
{
    fprintf(stderr, "send failed");
    exit(1);
}
```

最后将生成的服务器响应发送给客户端即可。

2. HTTPS_SERVER

基本逻辑与 HTTP_SERVER 类似，使用 OpenSSL 库来进行加密通信。本人选择了 TLSv1.2 作为 SSL/TLS 协议版本，首先创建了一个 SSL 上下文（SSL_CTX）对象，并加载了证书和私钥。

```
SSL_library_init();
OpenSSL_add_all_algorithms();
SSL_load_error_strings();

const SSL_METHOD *method = TLSv1_2_server_method();
SSL_CTX *ctx = SSL_CTX_new(method);

// load certificate and private key
if (SSL_CTX_use_certificate_file(ctx, "./keys/cnlab.cert", SSL_FILETYPE_PEM) <= 0)
{
    perror("load cert failed");
    exit(1);
}
if (SSL_CTX_use_PrivateKey_file(ctx, "./keys/cnlab.prikey", SSL_FILETYPE_PEM) <= 0)
{
    perror("load prikey failed");
    exit(1);
}
```

之后与 HTTP 服务器类似，建立 socket 文件描述符，然后绑定监听地址，最后对 443 端口进行监听。然后是依旧是循环不断接收连接请求，在每收到一个服务器的连接后，创建一个新的线程用于接收并解析其请求，然后根据请求内容进行应答。

```
while (1)
{
    struct sockaddr_in c_addr;
    socklen_t addr_len;

    int request = accept(sockfd, (struct sockaddr *)&c_addr, &addr_len);
    if (request < 0)
    {
        perror("Accept failed");
        exit(1);
    }

    SSL *ssl = SSL_new(ctx);
    SSL_set_fd(ssl, request);

    pthread_t new_thread;

    if ((pthread_create(&new_thread, NULL, (void *)handle_https_request, (void *)ssl)) != 0)
    {
        perror("Create handle_http_request thread failed");
        exit(1);
    }
}
```

https 请求的处理函数的功能描述如下：

首先第一步是执行 SSL 握手，然后由于客户端可能需要继续保持连接，这里设置了一个变量 (keep-alive) 用于处理需要继续保持连接的情况 (其实就是一个 while 循环)，之后开始接受客户端的 request 报文，并检查是否为 GET 请求，然后对请求进行解析，得到请求的 URL、HTTP 版本以及是否有范围请求 (Range) 和连接保持 (Connection) 等信息。而根据解析的信息，确定要发送的文件路径。如果文件不存在，发送一个 404 错误响应，否则看是否有 Range 字段，如果有 Range 字段，返回 206 Partial Content，若没有会返回 200 OK。而如果文件存在，会根据请求返回文件的对应部分的内容。下面是相关代码的展示（与 HTTP 服务器类似的部分就不作展示了）：

```
SSL *ssl = (SSL *)arg;
if (SSL_accept(ssl) == -1)
{
    perror("SSL_accept failed");
    exit(1);
}
```

上图实现 SSL 的握手

```
if(iterator = (strstr(recv_buff, "Range:")))
{
    iterator += 13;
    range = 1;

    range_begin = 0;
    while(*iterator >= '0' && *iterator <= '9')
    {
        range_begin = range_begin * 10 + (*iterator) - '0';
        iterator++;
    }
    iterator++;

    if(*iterator < '0' || *iterator > '9')
    {
            range_end = -1;
    }
    else{
        range_end = 0;
        while(*iterator >= '0' && *iterator <= '9')
        {
            range_end = range_end * 10 + (*iterator)-'0';
            iterator++;
        }
    }
}
```

上图实现了对 Range 字段的解析。

```
if(iterator = (strstr(recv_buff, "Connection:")))
{
    iterator += 12;
    if(*iterator == 'k'){
        keep_alive = 1;
    }
    else if(*iterator == 'c'){
        keep_alive = 0;
    }
}
```

```
strcat(response,"\r\nConnection: ");
if(keep_alive)
    strcat(response, "keep-alive");
else
    strcat(response, "close");
```

上图是根据 request 请求中的 Connection 字段判断是否需要继续保持连接（keep_alive 是循环的判断条件）

```
file_path[0] = '.';
file_path[1] = '\0';
if(relative_url){
    strcat(file_path, temp_url);
}
else
{
    i = 0;
    int count = 3;
    while(count){
        if(temp_url[i] == '/'){
            count--;
        }
        i++;
    }
    strcat(file_path, temp_url + i);
}
```

上图考虑了是否为相对路径，获取文件路径（temp_url 的获取逻辑与 HTTP 服务器类似，这里不作展示了）

```
if(fp == NULL)
{
    memset(send_buff, 0, 6000);
    strcat(send_buff, http_version);
    strcat(send_buff, " 404 Not Found\r\n\r\n\r\n\r\n");
    SSL_write(ssl, send_buff, strlen(send_buff));

    break;
}
else
{
    char header[200] = {0};
    strcat(header,http_version);

    if(range){
        strcat(header, " 206 Partial Content\r\n");
    }
    else{
        strcat(header, " 200 OK\r\n");
    }
```

上图是根据文件是否存在以及是否有 Range 字段返回不同的状态码

```
int size,begin;
if(range){
    if(range_end==-1){
        fseek(fp,0L,SEEK_END);
        size = ftell(fp) - range_begin + 1;
        begin = range_begin;
    }
    else{
        size = range_end - range_begin + 1;
        begin = range_begin;
    }
}
else{
    fseek(fp,0L,SEEK_END);
    size = ftell(fp);
    begin = 0;
}
```

上图是根据 range 字段的范围，获取文件的对应内容

```
strcat(header, "Content-Length: ");
fseek(fp,begin,0);

char str_size[64] = {0};
sprintf(str_size, "%d", size);

char response[size + 200];
memset(response,0, size + 200);
strcat(response, header);
strcat(response, str_size);

strcat(response,"\r\nConnection: ");
if(keep_alive)
    strcat(response, "keep-alive");
else
    strcat(response, "close");

printf("%s\n", response);
strcat(response, "\r\n\r\n");
fread(&(response[strlen(response)]), 1, size, fp);
SSL_write(ssl,response,strlen(response));
```

上图根据之前对 request 请求的解析完成对响应头部信息的填充（Content-Length 和 Connection 字段），最后把文件对应内容填充到响应中，并返回给客户端。

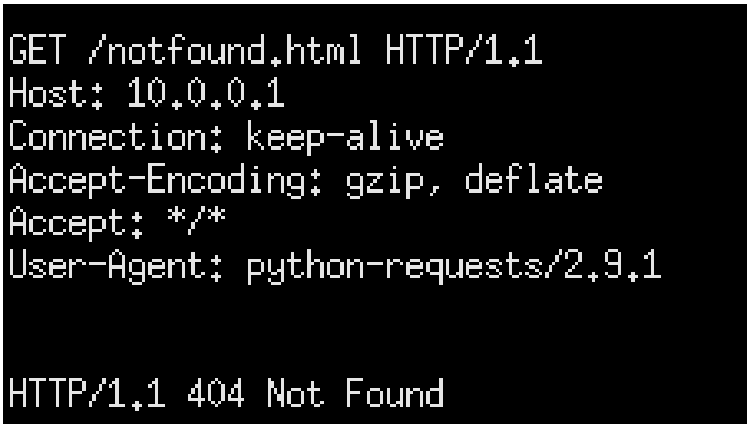
（二）运行结果展示

在代码中，将 request 和返回的服务器响应头部信息打印到控制台中，结果如下：

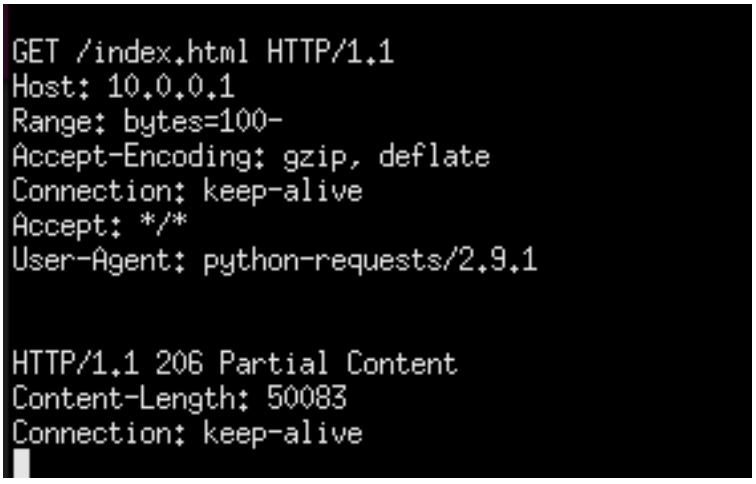
```
GET /index.html HTTP/1.1
Host: 10.0.0.1
Connection: keep-alive
Accept-Encoding: gzip, deflate
Accept: */*
User-Agent: python-requests/2.9.1

HTTP/1.1 301 Moved Permanently
Location: https://10.0.0.1/index.html
```

可见，当请求时 http 请求时，服务器返回 301 Moved Permanently，且在

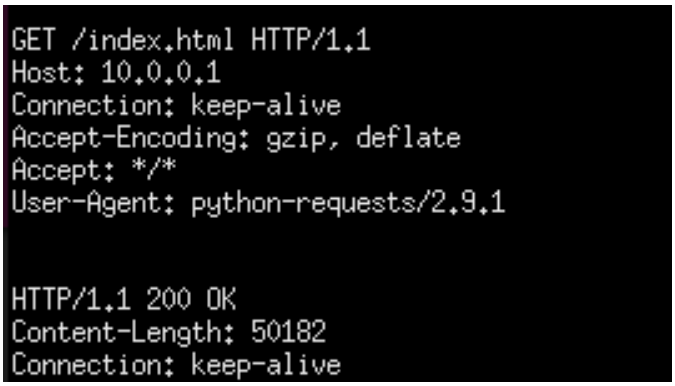
Location 字段中包含对应的 URL



当为 https 请求且文件不存在时，服务器返回 404 Not Found



当为 https 请求，文件存在时，若有 Range 字段，服务器返回 206 Partial Content。而文件大小，是否保持继续连接也与 request 对应



当为 https 请求，文件存在时，若没有 Range 字段，服务器返回 200 OK。而文件大小，是否保持继续连接也与 request 对应

五、 实验总结

通过本次实验,我对 Socket API 有了一定的了解。Socket API 对上层提供统一的调用接口,提供最基本的网络通信功能。通过实际编写一个简单的 HTTP 服务器,我对于建立 Socket 描述符、建立连接等内容以及 HTTPS 加密通信都有了不少了解，对于客户端、服务器之间的交互机制也有了一定的认识。此外我也学到了 HTTP 协议的内容,主要是对 HTTP 报文的结构有了不少了解,对 HTTPS 与 HTTP 的区别也有了更深的认识，因此本次实验让我对网络文件传输有了更深的理解。