二 Socket 应用编程实验

1 实验目的

使用 C 语言实现最简单的 HTTP 服务器，使用两个线程分别同时监听支持 HTTP（80 端口）和 HTTPS（443 端口）。只需支持 GET 方法，解析请求报文，返回相应应答及内容。需要支持以下状态码：

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

图 1. 需要支持的状态码

2 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：

```
./http-server
```

4. 在主机 h2 上运行测试程序，验证程序正确性：

```
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
```

5. 代码提交到 OJ 网站，报告提交到课程网站。

3 实验分析

在 main 函数中创建两个线程，分别监听 80 和 443 端口：

```
int main(){
   pthread_t http, https;
```

```
    if (pthread_create(&http, NULL, http_server, NULL) != 0){
        perror("HTTP thread creation failed");
        return -1;
    }
    if (pthread_create(&https, NULL, http_server, NULL) != 0){
        perror("HTTPS thread creation failed");
        return -1;
    }
    pthread_join(http, NULL);
    pthread_join(https, NULL);
    return 0;
}
```

在 http_server 函数中，创建 socket 并绑定到对应端口，然后监听请求。我是用一个循环来不断接收请求，再使用多线程同时处理请求，根据请求内容返回对应的应答：

```
void* http_server(void* arg){
    int port = 80;
    int sock;
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        perror("HTTP socket creation failed");
        exit(1);
    }

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(port);

    if (bind(sock, (struct sockaddr*)&addr, sizeof(addr)) < 0){
        perror("HTTP bind failed");
        exit(1);
    }
    listen(sock, 128);

    while (1)
    {
        struct sockaddr_in caddr;
        socklen_t addrlen;
        int request = accept(sock, (struct sockaddr*)&caddr, &addrlen);
        if (request < 0){
            perror("HTTP accept failed");
            exit(1);
        }
```

```
        pthread_t http_new_thread;
        if (pthread_create(&http_new_thread, NULL, (void*)handle_http_request,
            (void*)&request) != 0){
            perror("HTTP handle thread creation failed");
            exit(1);
        }
    }

    close(sock);
    return NULL;
}
```

上面这段代码中，socket 函数用于创建 socket，sockaddr_in 结构体用于存储地址信息，bind 函数用于将 socket 绑定到端口，listen 函数用于监听请求，accept 函数用于接收请求，pthread_create 函数用于创建每一个请求的处理线程，close 函数用于关闭 socket。

在每一个请求的处理线程中，首先接收到请求报文，然后解析请求报文，根据请求内容返回对应的应答：

```
void* handle_http_request(void* arg){
    pthread_detach(pthread_self());
    int request = *(int*)arg;

    char* recv_buff = (char*)malloc(2000 * sizeof(char));
    char* send_buff = (char*)malloc(6000 * sizeof(char));
    memset(recv_buff, 0, 2000);
    memset(send_buff, 0, 6000);

    int request_len = recv(request, recv_buff, 2000, 0);
    if (request_len < 0){
        fprintf(stderr, "HTTP receive failed");
        exit(1);
    }

    char *get = strstr(recv_buff, "GET");
    if (get){
        char *pos = get + 4; // skip "GET "
        char *url = (char*)malloc(50 * sizeof(char));
        char *http_version = (char*)malloc(9 * sizeof(char));
        char *host = (char*)malloc(100 * sizeof(char));
        int relative_url = *pos == '/'; // relative url has a leading '/'

        int i = 0;
        for (i = 0; *pos != ' '; pos++, i++)
```

```
            url[i] = *pos;
        url[i] = '\0';
        pos++;

        for (i = 0; *pos != '\r'; pos++, i++)
            http_version[i] = *pos;
        http_version[i] = '\0';

        if (relative_url){
            pos = strstr(recv_buff, "Host:"); // find Host field
            if (!pos){
                fprintf(stderr, "HTTP Host not found");
                exit(1);
            }
            pos += 6; // skip "Host: "

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


        strcat(send_buff, host);
        strcat(send_buff, http_version);
        strcat(send_buff, " 301 Moved Permanently\r\nLocation");
        strcat(send_buff, "https://");

        if (relative_url){
            strcat(send_buff, host);
            strcat(send_buff, url);
        }
        else
            strcat(send_buff, &url[7]); // skip "http://"
        strcat(send_buff, "\r\n\r\n\r\n\r\n"); // end of response

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

        free(url);
        free(http_version);
        free(host);
```

```
    }
    free(send_buff);
    free(recv_buff);

    close(request);
    return NULL;
}
```

上面这段代码中，我先使用 pthread_detach 函数将线程设置为分离状态，然后使用 recv 函数接收请求报文，存入缓冲区中，再使用 strstr 函数找到请求报文中的 GET 方法。这里我区分了绝对 URL 和相对 URL，对于绝对 URL，直接返回 301 状态码和路径，对于相对 URL，需要找到 Host 字段，再返回 301 状态码，拼接之后得到绝对路径。最后使用 send 函数发送应答报文，关闭 socket。

https_server 函数与 http_server 较为类似，区别在于它使用 Open SSL 库进行加密通信，需要先加载证书和私钥：

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

const SSL_METHOD *method = TLS_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);
}
```

之后的步骤就大体一致了，建立 socket，绑定端口，监听请求，使用循环不断接收请求，为每一个请求创建一个处理线程，处理请求，返回应答。下面给出部分代码，省略了一些重复的部分：

```
while (1)
{
    struct sockaddr_in caddr;
    socklen_t addrlen;
    int request = accept(sock, (struct sockaddr*)&caddr, &addrlen);
    if (request < 0){
        perror("HTTPS accept failed");
        exit(1);
```

```
    }

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


    pthread_t https_new_thread;
    if (pthread_create(&https_new_thread, NULL, (void*)handle_https_request, (void*)&ssl)
        != 0){
        perror("HTTPS handle thread creation failed");
        exit(1);
    }
}
```

不同之处在于，这里需要创建 SSL 结构体，将 socket 绑定到 SSL 结构体，然后传入处理线程中。处理线程中，需要使用 SSL 结构体的函数来接收和发送数据：

```
void* handle_https_request(void* arg){
    pthread_detach(pthread_self());
    SSL *ssl = (SSL*)arg;
    if (SSL_accept(ssl) == -1){
        fprintf(stderr, "HTTPS SSL_accept failed");
        exit(1);
    }

    char* recv_buff = (char*)malloc(2000 * sizeof(char));
    char* send_buff = (char*)malloc(6000 * sizeof(char));
    int keep_alive = 1;
```

上面代码中，我使用 `SSL_accept` 函数接收请求，建立缓冲区，设置 `keep_alive` 变量，用于判断是否保持连接。

我仍然区分了绝对 URL 和相对 URL，在 https 的处理函数中，我先判断是否有 Range 字段和保持连接信息，在确定要发送的文件路径。如果文件不存在，返回 404 状态码，若存在且有 Range 字段，返回 206 状态码，若存在但无 Range 字段，返回 200 状态码，最后发送应答报文：

```
    while (keep_alive){
        memset(recv_buff, 0, 2000);
        int request_len = SSL_read(ssl, recv_buff, 2000);
        if (request_len < 0){
            fprintf(stderr, "HTTPS receive failed");
            exit(1);
        }
        if (recv_buff[0] == '\0')
```

```
            break;

        char *url = (char*)malloc(50 * sizeof(char));
        char *http_version = (char*)malloc(9 * sizeof(char));
        char *path = (char*)malloc(100 * sizeof(char));
        char *get = strstr(recv_buff, "GET");

        if (get){
            char *pos = get + 4;
            int relative_url = *pos == '/';
            int range = 0;
            int range_begin, range_end;

            int i = 0;
            for (i = 0; *pos != ' '; pos++, i++)
                url[i] = *pos;
            url[i] = '\0';
            pos++;

            for (i = 0; *pos != '\r'; pos++, i++)
                http_version[i] = *pos;
            http_version[i] = '\0';

            if (pos = strstr(recv_buff, "Range:")){
                pos += 13; // skip "Range: bytes="
                range = 1;
                range_begin = 0;

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

                if (*pos < '0' || *pos > '9')
                    range_end = -1; // -1 means to the end of file
                else{
                    range_end = 0;
                    while(*pos >= '0' && *pos <= '9'){
                        range_end = range_end * 10 + *pos - '0';
                        pos++;
                    }
                }
```

```
            }

            if (pos = strstr(recv_buff, "Connection:")){
                pos += 12;
                if (*pos == 'k') // keep-alive, otherwise close
                    keep_alive = 1;
                else
                    keep_alive = 0;
            }

            path[0] = '.';
            path[1] = '\0';
            if (relative_url)
                strcat(path, url);
            else{
                int count = 3;
                for (int i = 0; count > 0; i++){
                    if (url[i] == '/')
                        count--;
                }
                strcat(path, &url[i]);
            }


            FILE *fp = fopen(path, "r");
            if (!fp){
                memset(send_buff, 0, 6000);
                strcat(send_buff, http_version);
                strcat(send_buff, " 404 Not Found\r\n\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");

                int file_size, file_begin;
                if (range){
                    if(range_end == -1){
```

```
                    fseek(fp, 0L, SEEK_END); // move to the end of file
                    file_size = ftell(fp) - range_begin + 1; // calculate the size of file
                    file_begin = range_begin;
                }
                else{
                    file_size = range_end - range_begin + 1;
                    file_begin = range_begin;
                }
            }
            else{
                fseek(fp, 0L, SEEK_END);
                file_size = ftell(fp);
                file_begin = 0;
            }

            strcat(header, "Content-Length: ");
            fseek(fp, file_begin, 0);
            char file_size_str[64] = {0};
            sprintf(file_size_str, "%d", file_size); // convert file size to string

            char *response = (char*)malloc((file_size + 200) * sizeof(char));
            memset(response, 0, file_size + 200);
            strcat(response, header);
            strcat(response, file_size_str);

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

            strcat(response, "\r\n\r\n");
            fread(response + strlen(response), 1, file_size, fp);
            SSL_write(ssl, response, strlen(response));

            fclose(fp);
            if(range == 1 && range_end == -1)
                break;
        }
    }
    free(url);
    free(http_version);
    free(path);
```

```
    }
    free(recv_buff);
    free(send_buff);

    SSL_free(ssl);
    close(SSL_get_fd(ssl));
    return NULL;
}
```