中国科学院大学
《计算机网络（研讨课）》实验报告

姓名 寇逸欣 学号 2023K8009922004 专业 计算机科学与技术
实验项目编号 02 实验名称 socket 应用编程实验

一、 实验内容

本实验要求使用 C 语言实现一个简单的 HTTP 服务器，能够处理基本的 GET 请求，解析请求报文、返回相应应答及内容。具体要求如下：

表 1: 需要支持的状态码及场景

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

二、 实验流程

2.1 主函数设计

任务要求我们同时监听 80 和 443 端口，故而想到在主函数中创建两个线程，分别调用监听端口函数，传递参数 80 和 443。主函数代码如下：

```
int main()
{
    pthread_t http_thread;
    pthread_t https_thread;

    int http_port = HTTP_PORT;
    int https_port = HTTPS_PORT;

    if (pthread_create(&http_thread, NULL, listen_port, &http_port)) {
        perror("creat http thread error!\n");
        exit(1);
    }
    if (pthread_create(&https_thread, NULL, listen_port, &https_port)) {
        perror("creat https thread error!\n");
        exit(1);
    }

    pthread_join(http_thread, NULL);
```

```
    pthread_join(https_thread, NULL);

    return 0;
}
```

pthread_create 函数创建线程，传递的参数是监听端口函数 listen_port 和端口号。pthread_join 函数等待线程结束。

2.2 端口监听函数

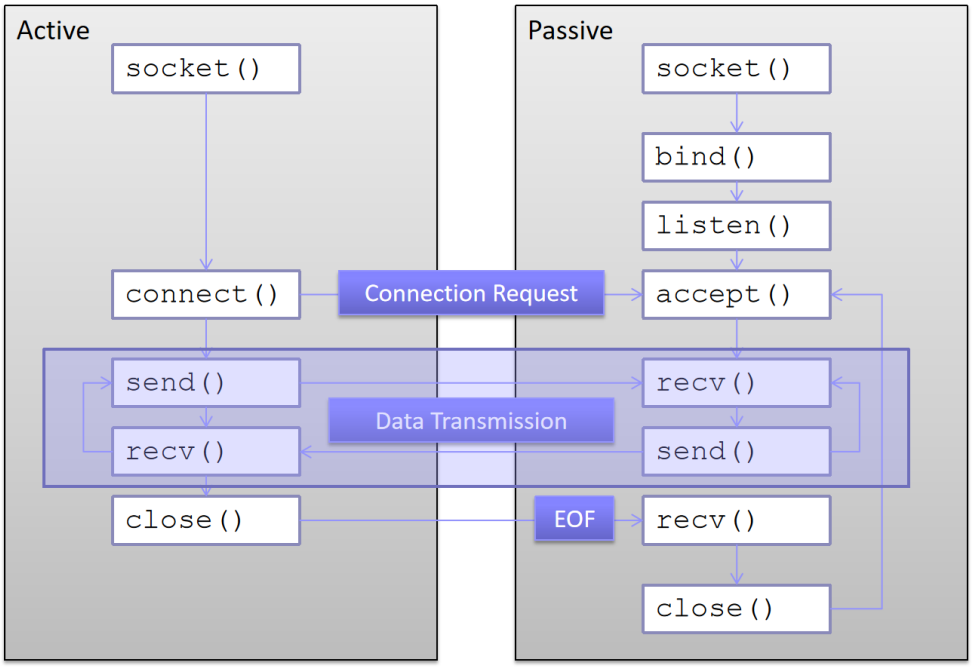

图 1: 客户端应答流程图

在给的代码框架中，已经有了这部分的代码，我们只需要修改一些细节。具体来说就是要设置一个端口号，用来区分是 80 端口还是 443 端口。然后在接收到请求后，调用处理请求函数 handle_https_request 或 handle_http_request。

2.3 request 解析

我们在接受到请求后，自然的想要将请求报文解析出来。故首先，我们需要设计数据结构 Request 来存储这些数据，包含请求方法、URL、版本号、Range 字段等。

```
typedef struct Header {
    char *name;
    char *value;
    struct Header *next;
} Header;

typedef struct line {
    char method[8];
```

```
    char url[256];
    char version[16];
} Line;

typedef struct Request{
    Line line;
    Header *headers;
    char *body;
} Request;
```

然后我们需要设计一个函数来解析请求报文。请求报文的样式如下：

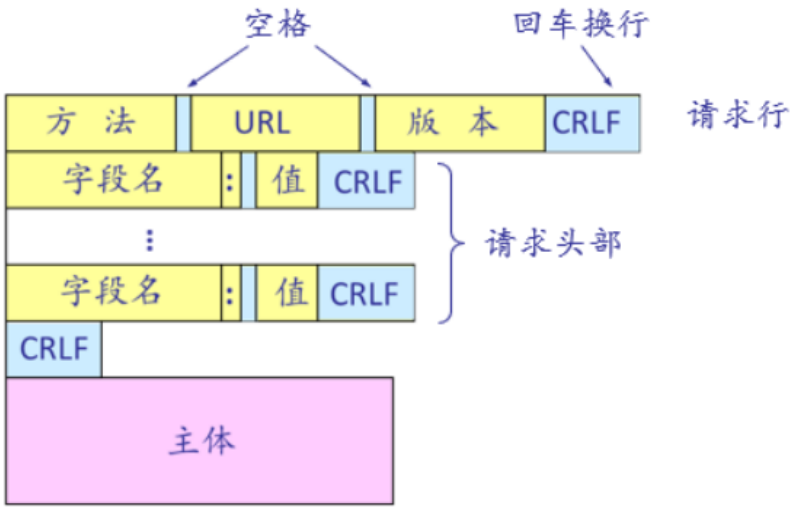

图 2: 请求报文格式

由此可知，我们可以先按行分割请求报文，然后解析第一行，按空格分割得到请求方法、URL 和版本号。接下来解析后续的每一行，按冒号分割得到 Header 的 name 和 value，存入链表中，然后是请求体。最后将解析结果存入 Request 结构体中。

```
void decode_request(char *raw_request, Request *request)
{
    char *line_end = strstr(raw_request, "\r\n");
    sscanf(raw_request, "%s %s %s", request->line.method, request->line.url, request->line.version);

    char *header_start = line_end + 2;
    char *header_end;
    Header *current_header = NULL;

    while ((header_end = strstr(header_start, "\r\n")) != NULL && header_end != header_start) {
        Header *new_header = (Header *)malloc(sizeof(Header));
        new_header->next = NULL;

        char *colon_pos = strstr(header_start, ": ");
        if (colon_pos != NULL) {
            int name_len = colon_pos - header_start;
```

```
            int value_len = header_end - (colon_pos + 2);

            new_header->name = (char *)malloc(name_len + 1);
            new_header->value = (char *)malloc(value_len + 1);

            strncpy(new_header->name, header_start, name_len);
            new_header->name[name_len] = '\0';
            strncpy(new_header->value, colon_pos + 2, value_len);
            new_header->value[value_len] = '\0';

            if (current_header == NULL) {
                request->headers = new_header;
            } else {
                current_header->next = new_header;
            }
            current_header = new_header;
        }

        header_start = header_end + 2;
    }

    if (strcmp(request->line.method, "POST") == 0) {
        request->body = strdup(header_start);
    } else {
        request->body = NULL;
    }
}
```

最终，h1 所收到并解析出的请求报文如下：

```
Method: GET, URL: /index.html, Version: HTTP/1.1
Header: Host: 10.0.0.1
Header: User-Agent: python-requests/2.31.0
Header: Accept-Encoding: gzip, deflate
Header: Accept: */*
Header: Connection: keep-alive

Method: GET, URL: /index.html, Version: HTTP/1.1
Header: Host: 10.0.0.1
Header: User-Agent: python-requests/2.31.0
Header: Accept-Encoding: gzip, deflate
Header: Accept: */*
Header: Connection: keep-alive

Method: GET, URL: /index.html, Version: HTTP/1.1
Header: Host: 10.0.0.1
Header: User-Agent: python-requests/2.31.0
Header: Accept-Encoding: gzip, deflate
Header: Accept: */*
Header: Connection: keep-alive

Method: GET, URL: /index.html, Version: HTTP/1.1
Header: Host: 10.0.0.1
Header: User-Agent: python-requests/2.31.0
Header: Accept-Encoding: gzip, deflate
Header: Accept: */*
Header: Connection: keep-alive

Method: GET, URL: /notfound.html, Version: HTTP/1.1
Header: Host: 10.0.0.1
Header: User-Agent: python-requests/2.31.0
Header: Accept-Encoding: gzip, deflate
Header: Accept: */*
Header: Connection: keep-alive

Method: GET, URL: /notfound.html, Version: HTTP/1.1
Header: Host: 10.0.0.1
Header: User-Agent: python-requests/2.31.0
Header: Accept-Encoding: gzip, deflate
```

图 3: h1 收到的请求报文

2.4 http 请求处理

对于 http 请求，我们永远将其重定向到 https。故而我们只需要返回 301 状态码，并在 Location 字段中拼接"https://10.0.0.1" 和请求的 URL 即可。

最终 http 发送的应答报文如下：

```
GET 301 Moved Permanently to https://10.0.0.1/index.html
GET 301 Moved Permanently to https://10.0.0.1/index.html
GET 301 Moved Permanently to https://10.0.0.1/notfound.html
GET 301 Moved Permanently to https://10.0.0.1/dir/index.html
GET 301 Moved Permanently to https://10.0.0.1/index.html
GET 301 Moved Permanently to https://10.0.0.1/index.html
```

图 4: h1 发送的 http 应答报文

2.5 https 请求处理

对于 https 请求，我们需要根据请求的 URL 来判断请求的文件是否存在，若存在则返回 200 状态码和文件内容，若不存在则返回 404 状态码。另外，我们还需要处理 Range 字段，若请求中包含 Range 字段，则返回 206 状态码和部分内容。

根据 test 文件可知，请求文件的 url 的格式为"/path/filename"，故我们只需要在前面加上"." 即可得到文件路径"./path/filename"。

对于 Range 字段，我们需要解析出起始位置和结束位置。不过结束位置可能不存在，若不存在则表示请求从起始位置到文件末尾的内容。

此时我们设置一个 file_pointer 指向文件的起始位置，然后使用 fseek 函数将文件指针移动到起始位置，接着使用 fread 函数读取指定长度的内容。最后将读取的内容存入响应报文中返回给客户端。

https 返回的应答报文如下：

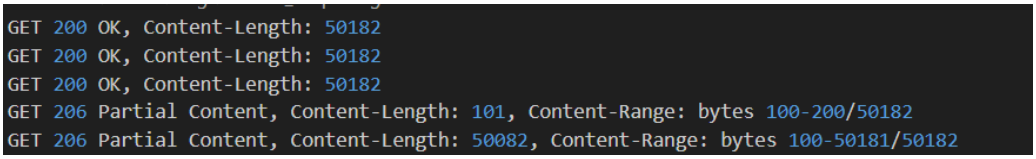

图 5: h1 发送的 https 应答报文

206 返回的结果如下图：

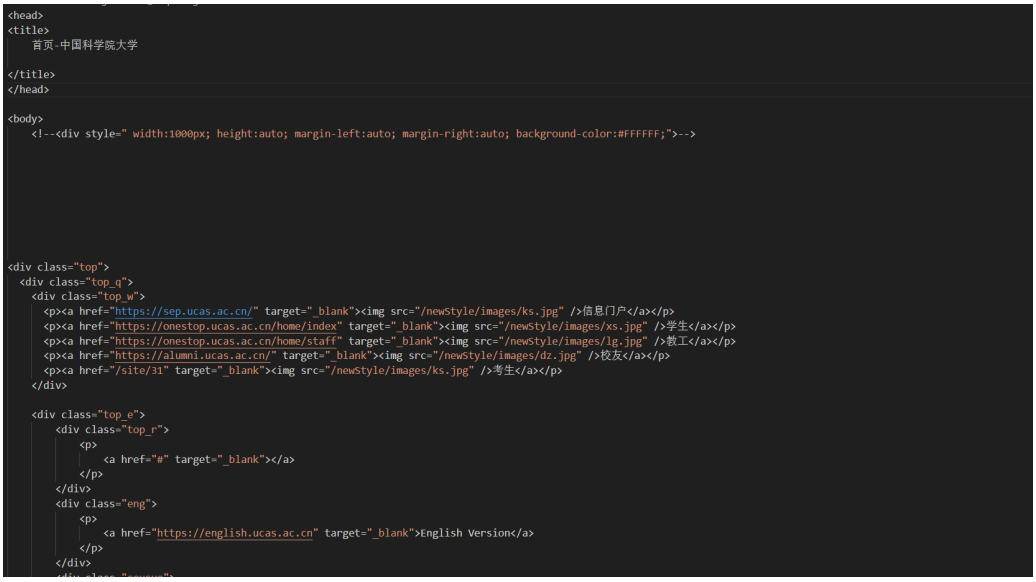

图 6: h1 发送的 206 应答报文

由于 https 和 http 请求处理函数大部分代码相似，我们这里只展示 https 请求处理函数 handle_https_request 的代码。

```
void handle_https_request(SSL* ssl)
{
    char *request = calloc(1024, sizeof(char));
    char *response = calloc(1024, sizeof(char));
    int request_len = 0;
    int response_len = 0;

    if (SSL_accept(ssl) == -1){
      perror("SSL_accept failed");
      exit(1);
   }

    request_len = SSL_read(ssl, request, 1024);
    if (request_len < 0) {
        perror("SSL_read failed");
        exit(1);
```

```
}

Request *http_request = calloc(1, sizeof(Request));
decode_request(request, http_request);

int option = 0; // 0: 200 OK, 1: 206 Partial Content
FILE *file_pointer = NULL;

for (Header *header = http_request->headers; header != NULL; header = header->next) {
    if (strcmp(header->name, "Range") == 0) {
        option = 1;
        break;
    }
}

// search file
char file_path[256] = ".";
char *file_path_pointer = file_path + 1;
strcat(file_path, http_request->line.url);

if ((file_pointer = fopen(file_path_pointer - 1, "r")) == NULL) {
    response_len = sprintf(response, "%s %d Not Found\r\n\r\n", http_request->line.version,
        NOT_FOUND);
    SSL_write(ssl, response, response_len);
} else {
    if (option == 0) {
        // 200 OK
        fseek(file_pointer, 0, SEEK_END);
        int file_size = ftell(file_pointer);
        fseek(file_pointer, 0, SEEK_SET);

        response_len = sprintf(response, "%s %d OK\r\nContent-Length: %d\r\n\r\n",
            http_request->line.version, OK, file_size);
        SSL_write(ssl, response, response_len);

        char *file_buffer = (char *)malloc(file_size);
        fread(file_buffer, 1, file_size, file_pointer);
        SSL_write(ssl, file_buffer, file_size);
        free(file_buffer);
    } else if (option == 1) {
        // 206 Partial Content
        char range_value[64];
        for (Header *header = http_request->headers; header != NULL; header = header->next) {
            if (strcmp(header->name, "Range") == 0) {
                strcpy(range_value, header->value);
                break;
            }
        }

        int start, end;
```

```
            if (sscanf(range_value, "bytes=%d-%d", &start, &end) != 2) {
                sscanf(range_value, "bytes=%d-", &start);
                end = -1;
            }

            fseek(file_pointer, 0, SEEK_END);
            int file_size = ftell(file_pointer);
            if (end == -1 || end >= file_size) {
                end = file_size - 1;
            }
            int content_length = end - start + 1;
            fseek(file_pointer, start, SEEK_SET);

            response_len = sprintf(response, "%s %d Partial Content\r\nContent-Length:
                %d\r\nContent-Range: bytes %d-%d/%d\r\n\r\n", http_request->line.version,
                Partial_Content, content_length, start, end, file_size);
            SSL_write(ssl, response, response_len);

            char *file_buffer = (char *)malloc(content_length);
            fread(file_buffer, 1, content_length, file_pointer);
            SSL_write(ssl, file_buffer, content_length);
            free(file_buffer);
        }
        fclose(file_pointer);
    }

    return;
}
```

三、 遇到的问题

1. 在实现 https 请求处理函数时，最开始没有考虑到 Range 结束位置可能不存在的情况，导致程序在处理类似”Range: bytes=200-” 的请求时出错。后来通过检查 sscanf 的返回值来解决这个问题。

2. 在实现 http 请求处理函数时，最开始没有正确设置 Location 字段，导致无法正确重定向到 https。后来通过在 Location 字段中拼接”https://10.0.0.1” 来解决这个问题。