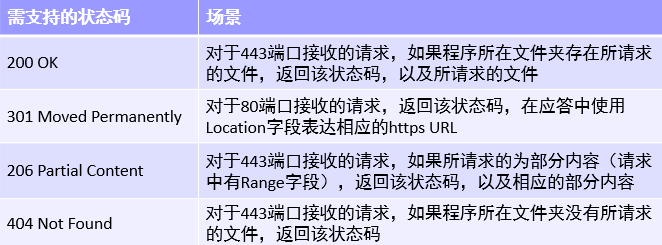
**Socket应用编程实验**

1. **实验内容**

实现：使用C语言实现最简单的HTTP服务器

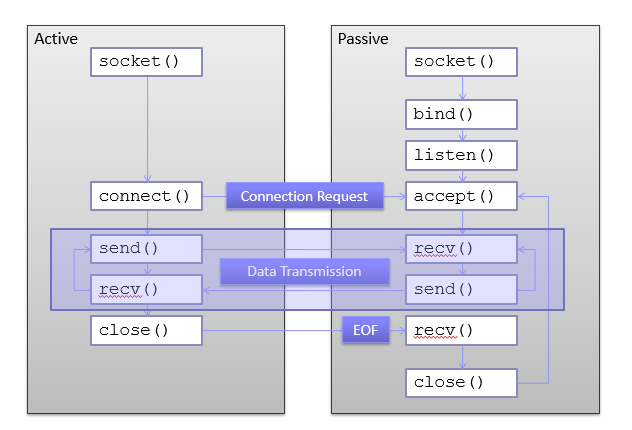
* 1. 同时支持HTTP（80端口）和HTTPS（443端口），使用两个线程分别监听各自端口

2、只需支持GET方法，解析请求报文，返回相应应答及内容。其中应答具体要求如下。



图表 1：状态码

1. **设计思路**
2. 提供的实验代码实现了一个基本的HTTPS服务器，该代码实现443端口和部分200状态码的功能实现。
3. 基于实验内容，要在现有代码的框架实现双线程，增加80端口和返回301状态码的代码，在443端口增加支持206和404状态码返回的代码。



图表 2：客户端应答流程

整体HTTP和HTTPS服务器设计思路如上图所示，我们只需完成接收和返回部分。

3、具体实现

（1）双线程

在main函数中创建两个线程，其中创建thread\_1进程用于监听HTTP端口，主线程监听HTTPS端口。两个线程都使用listen\_port函数进行监听。最后pthread\_detach函数在第一个线程结束后自动释放资源，

int main(){

    pthread\_t thread\_1;

    int http\_port = HTTP;

    int https\_port = HTTPS;

    if(pthread\_create(&thread\_1,NULL,listen\_port,&http\_port) != 0){

        perror("Create thread failed");

        exit(1);

    }

    listen\_port(&https\_port);

    pthread\_detach(thread\_1);

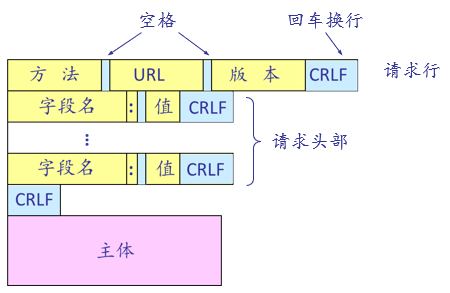
}

（2）配置服务器监听的地址和端口

这部分与提供代码main函数部分大致相同，不做赘述。

（3）request解析

这一部分使用reques\_decode函数对request进行解析，将函数分为method，url，version，头部和主体几个部分，这样在后续处理中针对状态码可以直接应用，比较方便。函数中主要使用指针作为标志位，标志处理的位置。request大致如下图所示。



图表 3：请求样式

在实际中，请求如下图所示



图表 4：实际请求样式

struct Request\* request\_decode(const char\* raw\_request)

raw\_request指向request中未被处理的第一位。

首先，定义一个指向request的结构体指针req，在该函数中会将该结构体中所有变量赋值。

struct Request \*req = NULL;

typedef struct Request {

    enum Method method;

    char \*url;

    char \*version;

    struct Header \*headers;

    char \*body;

} Request;

其次，判断是否为GET方法，使用strcspn判断长度，memcpy函数进行类型判断

size\_t method\_len = strcspn(raw\_request, " ");

    if (memcmp(raw\_request, "GET", strlen("GET")) == 0) {

        req -> method = GET;

    }

    else {

        req -> method = UNSUPPORTED;

    }

    raw\_request += (method\_len + 1);

url和version部分同样使用strcspn函数，同时使用memcpy函数将需要的部分放进req结构体中，然后移动raw\_request指针。

 size\_t url\_len = strcspn(raw\_request, " ");

    req -> url = malloc(url\_len + 1);

    memcpy(req -> url, raw\_request, url\_len);

    req -> url[url\_len] = '**\0**';

    raw\_request += url\_len + 1; // move past <SP>

而后处理header部分，这其中对于状态码有用的部分是host和range字段，但在这里我以“：“为标志位，将所有name和后面的内容拆分出来用链表连接。如下图为一个链表，识别和赋值方法与url类似。

typedef struct Header {

    char \*name;

    char \*value;

    struct Header \*next;

} Header;

之后处理body部分，将所有剩余字符赋值给req->body。

size\_t body\_len = strlen(raw\_request);

    req -> body = malloc(body\_len + 1);

    memcpy(req -> body, raw\_request, body\_len);

    req -> body[body\_len] = '**\0**';

（4）HTTP端口处理

struct Request\* req = request\_decode(request\_buf);

    // 301

    code = MOVED\_PERMANTLY;

    char new\_url[100];

    strcpy(new\_url, "https://10.0.0.1");

    strcat(new\_url, req->url);

    response\_len += sprintf(response\_buf, "%s %d MOVED PERMANENTLY**\r\n**", req->version, code);

    response\_len += sprintf(response\_buf + response\_len, "Location: %s**\r\n**", new\_url);

从request\_decode函数中获得分解后的请求结构体后，将状态码301需要的部分放进response\_buf即可。返回内容如下图所示。



图表 5：应当输出

（5）HTTPS端口处理

由于HTTPS有三个状态码需要处理，因此需要进行判断。判断依次为是否打开文件成功和是否返回部分文件。

如果状态码为404的话，打开文件失败，将版本和状态码放入应答缓冲区即可。

if ((fp = fopen(refined\_url, "r")) == NULL) {

        code = NOT\_FOUND;

        response\_len += sprintf(response\_buf, "%s %d NOT FOUND**\r\n**", req -> version, code);

        response\_len += sprintf(response\_buf + response\_len, "**\r\n**");

    }

而后进行状态码206和状态码200的判断，在头部查找range字段，如果找到则设置partial为一，即状态码为206。使用sscanf函数对start和end进行赋值，如果range字段形如“100-”则end会赋值为file\_len-1，即文件末尾

int start = 0;

        int end = file\_len - 1;

        char str\_range[15];

        strcpy(str\_range, "Range");

        for (h = req -> headers; h; h = h -> next) {

            if (strcmp(h -> name, str\_range) == 0) {

                partial = 1;

                sscanf(h -> value, "bytes=%d-%d", &start, &end);

                end += 1;

                break;

            }

        }

而之后对于状态码206和200的处理类似，这里只叙述我是如何处理206的。先使用fseek将文件指针移动到请求开始的位置，而后在应答缓冲区中放入版本、状态码和内容长度。使用fread读取需要的文件内容，放入缓冲区中即可。

            code = PARTIAL\_CONTENT;

            int read\_len = end - start;

            fseek(fp, start, SEEK\_SET);

            response\_len += sprintf(response\_buf, "%s %d PARTIAL CONTENT**\r\n**", req -> version, code);

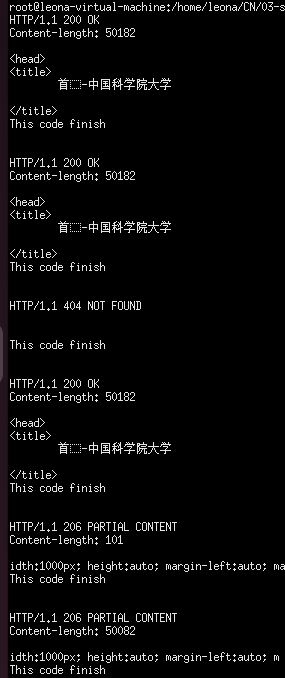
            response\_len += sprintf(response\_buf + response\_len, "Content-length: %d**\r\n**", read\_len);

            response\_len += sprintf(response\_buf + response\_len, "**\r\n**");

            fread(response\_buf + response\_len, sizeof(char), read\_len, fp);

            fseek(fp, 0, SEEK\_SET);

            response\_len += read\_len;

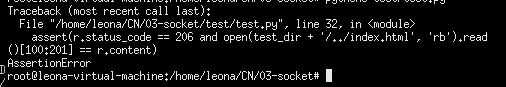


图表6：只输出前一百个字符

1. **遇到的问题**
   1. request\_decode中解析请求

这个函数使用raw\_request指针进行请求的处理，但是如何移动使其准确指向未被处理的第一个字符是一个比较大的问题。

h2窗口中出现了报错。



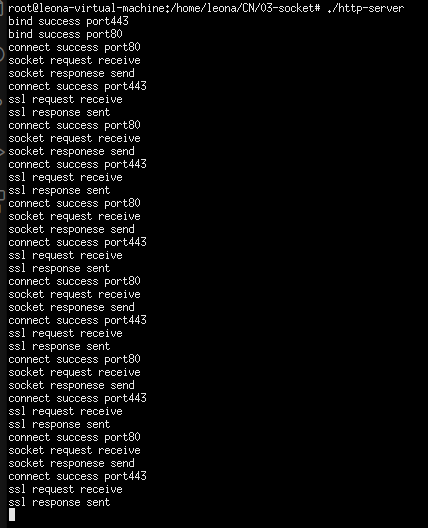
图表 6：h2窗口报错

在h1窗口中查看所有请求信息，下图状态码的应带中url出现问题且状态码出现错位，考虑请求的请求行出现问题，进行检查后，发现raw\_request处理版本移位出现错误，由于版本后接“\r\n”，因此不能像url部分一样长度+1（raw\_request += url\_len + 1），而是应该raw\_request += version\_len + 2，改正后输出正常。



图表 7：h1窗口错误输出

1. **测试**



图表 7：将请求和应答均打印出来的结果



图表 8：将80端口和443端口应答均打印