HTTP-SERVER 实验报告

小组成员：郭宇祺　卫思为

1　实验内容简介

我们的 HTTP 服务器实现了三个基本功能：服务器主页获取、文件上传、文件下载。我们在服务器主页获取和文件下载这两个功能上实现了 HTTP GET 方法，在文件上传这个功能上实现了 HTTP POST 方法以及分块传输功能。同时我们针对文件上传功能实现了 HTTP pipeline。我们还借助 openssl 库实现了 SSL 加密，目前我们的服务器只能通过 https 协议访问，不能通过普通 http 协议访问。

2　实现方法

2.1　HTTP POST/GET 方法

在本次实验中，由于我们的 http-server 使用纯 C 语言编写，因此无法使用高级框架解析请求头，也很难使用高级数据结构存储解析得到的信息。因此，我们使用较为简单的字符串匹配的方法手动实现 HTTP 请求头的解析。在我们的实现中，我们所实现的 HTTP 请求头解析功能只会解析与本次实验内容相关的项，对于其他无关项，我们的请求头解析功能不做识别和处理。

HTTP GET 方法的实现较为简单。在 HTTP 请求头中，第一行就包含了本次 HTTP 请求所访问的 URI 信息。在 http 协议中，对于 URI，使用符号”?” 作为分隔符分隔访问路径与参数，”?” 之前的字符串表示所要访问的文件/应用所在路径，之后的字符串表示访问对应文件/应用时所需要的参数。参数以键值对 key=value 的形式出现，多个参数之间使用”&” 符号进行分隔。基于以上原则，我们编写代码实现了 HTTP GET 方法的解析功能，相关代码主要集中在 lib/server_handler.c 文件中的 cut_params() 函数和 get_value() 函数中。需要注意的是，由于 URI 可能包含非英文字符，在使用之前需要先进行解码，解码代码位于 lib/safe_connect.c 文件中的 urldecode() 函数中。

HTTP POST 方法的实现略微复杂。与 HTTP GET 方法不同，HTTP POST 方法将请求参数放置于请求头的请求项中。对于文件上传功能，请求头会在 Content-Length 项中标识后续请求长度，并在 boundary 项中标识包裹文件内容所使用的特殊字符串。在每一个被 boundary 所指示的字符串所包裹的块中，都存在一个 filename 项，其指定了当前块内容所对应的文件名。根据以上信息，我们编写了 HTTP POST 解析功能，相关代码主要集中在 lib/server_handler.c 文件中的 get_value() 函数中。

2.2　文件上传、下载和分块传输

在服务器主页中，我们创建了如下的表单，以供用户选择文件并上传：

```
<form action="/upload" method="post" enctype="multipart/form-data">
上传文件：  <input type="file" name="upload_filename"><br>
<input type="submit">
</form>
```

当用户点击上传按钮后，浏览器会创建 POST 请求向服务器传送文件内容。我们的服务器在完成请求头解析、文件内容接受等一系列内容后，将接收到的文件保存到 resources 文件夹中，同时向浏览器返回重定向信息，将

页面重新导向服务器主页。并在页面最上端显示“xx 文件上传成功”的提示信息。文件上传的处理函数位于 lib/server_handler.c 文件中的 file_upload() 函数中。

同时，我们还在服务器主页中，添加了服务器上已有文件的信息，并将每一条信息做成如下所示的超链接：

```
<a href=\"/download?filename=${FILENAME}\">${FILENAME}</a><br>
```

每当用户点击对应的文件名，浏览器就会自动向服务器发送请求，开始文件的下载功能。我们的服务器在返回的请求头中设置如下两项，以便触发浏览器的下载功能：

```
Content-Type: application/octet-stream
Content-Disposition: attachment;filename=${FILENAME}
```

文件下载的处理函数位于 lib/server_handler.c 文件中的 file_download_chunked() 函数中。

TODO：分块传输

2.3 HTTP 持久连接和管道

2.4 使用 openssl 库实现 HTTPS

2.5 使用 libevent 实现多路并发

我们首先创建一个事件 listen_fd，指定其监视服务器套接字 server_fd，并设置其在接收到新连接时的回调函数为 on_accept()，相关代码如下:

```
struct event listen_ev;
base = event_base_new();
event_set(&listen_ev, server_fd, EV_READ | EV_PERSIST, on_accept, NULL);
event_base_set(base, &listen_ev);
event_add(&listen_ev, NULL);
event_base_dispatch(base);
```

server.c

随后，在函数 on_accept() 中，我们为服务器建立与客户端之间的连接，并处理相关请求，相关代码如下：

```
void on_accept(int server_fd, short event, void *arg)
{
  struct sockaddr_in client_addr;
  socklen_t client_addr_size = sizeof(client_addr);
  int client_fd;
  char recv_buffer[DEFAULT_RECV_BUFFER_SIZE];
  int n;
  char reqs[N_REQ][DEFAULT_RECV_BUFFER_SIZE] = {0};
  int recv_rest = 0;
  // read_ev must allocate from heap memory, otherwise the program would crash from segmant fault
  if ((client_fd = accept(server_fd, (struct sockaddr *)&client_addr,
              &client_addr_size)) == -1)
  {
    perror("accept failed:");
    return;
  }

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

  if (SSL_accept(ssl) <= 0)
  {
    perror("ssl state:");
  }
  memset(recv_buffer, 0, sizeof(char) * DEFAULT_RECV_BUFFER_SIZE);
  while (1)
  {
    if (n == 0)
      n = recv_s(ssl, recv_buffer + recv_rest, DEFAULT_RECV_BUFFER_SIZE - recv_rest, 0);
    if (n == 0)
      break;

```

```
    int n_buffer;
    int req_len[N_REQ] = {0};

    memset(reqs, 0, sizeof(char) * N_REQ * DEFAULT_RECV_BUFFER_SIZE);
    n_buffer = divide_buffer(recv_buffer, n, reqs, req_len, &recv_rest);

    for (int i = 0; i < n_buffer; i++)
    {
      handle(ssl, reqs[i], req_len[i]);
    }
    memmove(recv_buffer, recv_buffer + n - recv_rest, recv_rest);
    n = recv_rest;
  }
  SSL_shutdown(ssl);
  SSL_free(ssl);

  close(client_fd);
}
```

server.c

3　功能测试

3.1　文件上传下载测试

我们已经在 resources 文件夹中放置了一些文件，包含各种格式类型（pdf、jpg、pptx、docx、xlsx 等）。我们将这些文件上传后下载，并重新打开文件验证，发现这些文件的内容和格式都没有损坏。与此同时，服务器的主页也能够正确显示当前服务器上所存储的文件信息，如图1所示：

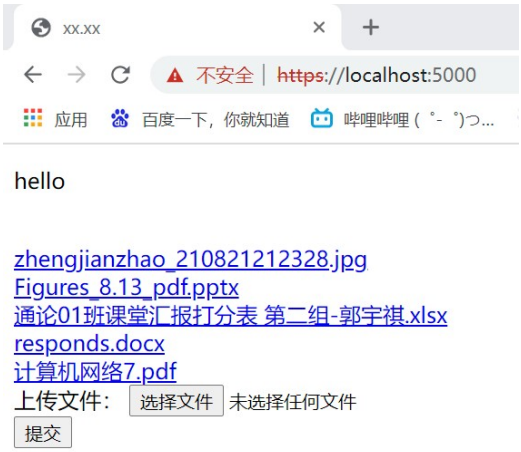

图 1: 服务器主页示意图

因此服务器的文件上传下载功能运行良好。

3.2　持久连接和管道测试

3.3　HTTPS 测试

3.4　多路并发测试