```c
//Name: Mitchell Larson
//Course: CE 4961
//Assignment: Project 4
//Description: This file servers as the main entry to a simple HTTP server.
//Note: Portions of this file were copied from Project 3 - echo client.

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>

#include "http_server.h"

int main(int argc, char** argv){

    //Define variables needed by application
    int server_sock_fd, client_sock_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len;
    int pid;

    //User should pass port to run HTTP server on
    if(argc != 2){
        printf("Usage: httpclient <port>\n");
        exit(1);
    }

    //Create a socket - Exit if unsuccessful
    if((server_sock_fd = socket(PF_INET, SOCK_STREAM, 0)) < 0){
        perror("Error creating socket");
        exit(1);
    }

    unsigned short port;

    //Get port from user - Exit with error code if unsuccessful
    if(sscanf(argv[1], "%hu", &port) != 1){
        perror("Error parsing port");
        exit(1);
    }

    //Add port info to server object
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(port);

    //bind to port provided by user
```

```
53      if(bind(server_sock_fd, (struct sockaddr*) &server_addr, sizeof(server_addr)) < 0){
54          perror("Error binding to port");
55          exit(1);
56      }
57
58      //Listen for TCP connection requests. Allowing 5 queued connections at a time
59      if(listen(server_sock_fd, 5) < 0){
60          perror("Error listening to socket");
61          exit(1);
62      }
63
64      //accept new connections in a seperate process
65      while(1){
66          client_addr_len = sizeof(client_addr);
67          client_sock_fd = accept(server_sock_fd, (struct sockaddr*) &client_addr, &client_addr_len);
68
69          if(client_sock_fd < 0){
70              perror("Error accepting client connection");
71          }
72
73          pid = fork();
74
75          if(pid < 0){
76              perror("Error forking process");
77          }else if(pid == 0){
78              //child process. Close main socket pipe.
79              close(server_sock_fd);
80              serve_client(client_sock_fd);
81              exit(0);
82          }else{
83              //parent process. Close pipe to child socket.
84              close(client_sock_fd);
85          }
86      }
87
88      return 0;
89  }
90
91  ////////////////////////////////////////////////////////////////////////////////////////////////////
92  ////////////////////////////////////////////////////////////////////////////////////////////////////
93  ////////////////////////////////////////////////////////////////////////////////////////////////////
94  /**
95   * @file    - http_server.c
96   * @author  - Mitchell Larson (larsonma@msoe.edu)
97   * @brief   - Serves an HTTP client by interperting request and fetching the
98   *            appropriate response.
99   * @version - 0.1
100  * @date    - 2019-01-17
101  *
102  * Copyright (c) 2019
103  *
104  */
```

```c

#include "http_server.h"

//webroot
static char root[6] = "./www";

static void reroute(struct HTTP_REQUEST_STUCT*, int);

/**
 * @brief - Serves an HTTP client by reading and writing to a TCP socket
 *
 * @param client_sock_fd - TCP socket file descriptor
 */
void serve_client(int client_sock_fd){
    struct HTTP_REQUEST_STUCT request = {};
    struct HTTP_RESPONSE_STRUCT response = {};
    FILE *inStream;

    //Open the file descriptor to use higher-level file I/O
    if(!(inStream = fdopen(client_sock_fd, "r"))){
        printf("Error opening TCP input stream\n");
    };

    //Parse the HTTP request parameters
    fscanf(inStream, "%s %s %s", request.header.HTTP_verb, request.header.url, request.header.version);

    //GET is the only supported command. Set error is not GET
    if(strcmp(request.header.HTTP_verb, "GET") != 0){
        response.header.status = 405;
    }else{
        strcpy(request.filepath, root);
        FILE *fp = fopen (strcat(request.filepath, request.header.url),"rb");

        printf("GET %s\n", request.filepath);

        //Check if the file exists and set status based on presence of file.
        //Bonus: set status to 418 if client attempts to brew coffee.
        if (fp == NULL && strcmp(request.header.url, "/brew/coffee") == 0) {
            response.header.status = 418;
        } else if(fp == NULL) {
            response.header.status = 404;
        }else {
            response.header.status = 200;
            fclose(fp);
        }
    }

    //If the file was not found, reroute so corresponding error file is returned.
    if(response.header.status != 200){
        reroute(&request, response.header.status);
    }

```

```
157         //Create the HTTP response based on parsed/derived parameters.
158         create_response(&request, &response);
159 
160         //Write the header to the client, followed by the body.
161         write(client_sock_fd, response.header_str, strlen((char*)response.header_str));
162         write(client_sock_fd, response.data, response.header.content_length);
163 
164         //Free dynamic data generated when creating HTTP response.
165         free(response.header_str);
166         free(response.data);
167 
168         //Close the connection.
169         close(client_sock_fd);
170     }
171 
172     /**
173      * @brief - This function reroutes an HTTP request so that the proper
174      *          HTML error page is displayed when an error occurs serving
175      *          the HTTP request.
176      *
177      * @param request
178      * @param status
179      */
180     void reroute(struct HTTP_REQUEST_STUCT *request, int status){
181         char newPath[16];
182         strcpy(newPath, root);
183 
184         switch(status){
185             case 404:
186                 strcpy(request->filepath, strcat(newPath, "/404.html\0"));
187                 break;
188             case 405:
189                 strcpy(request->filepath, strcat(newPath, "/405.html\0"));
190                 break;
191             case 418:
192                 strcpy(request->filepath, strcat(newPath, "/418.html\0"));
193                 break;
194             case 500:
195             default:
196                 strcpy(request->filepath, strcat(newPath, "/418.html\0"));
197                 break;
198         }
199     }
200 
201     ////////////////////////////////////////////////////////////////////////////////////////////
202     ////////////////////////////////////////////////////////////////////////////////////////////
203     ////////////////////////////////////////////////////////////////////////////////////////////
204     /**
205      * @file     - http_response.c
206      * @author   - Mitchell Larson (larsonma@msoe.edu)
207      * @brief    - Parses HTTP request and response structs to generate a valid
208      *             HTTP response.
```

```
 * @version  - 0.1
 * @date     - 2019-01-17
 *
 * Copyright (c) 2019
 *
 */

#include "http_response.h"

static void read_data(struct HTTP_REQUEST_STUCT*, struct HTTP_RESPONSE_STRUCT*);
static void set_header_data(struct HTTP_REQUEST_STUCT*, struct HTTP_RESPONSE_STRUCT*);
static void get_message(int, char*);
static void set_content_type(char*, struct HTTP_RESPONSE_HEADER*);
static void set_content_len(char*, struct HTTP_RESPONSE_HEADER*);
static void set_date(struct HTTP_RESPONSE_HEADER*);
static void set_hostname(struct HTTP_RESPONSE_HEADER*);
static void print_header(struct HTTP_RESPONSE_STRUCT* response);

/**
 * @brief  - Create a HTTP response by setting header data and file data.
 *
 * @param request - HTTP_REQUEST_STRUCT pointer containing request data.
 * @param response - HTTP_RESPONSE_STRUCT pointer containing parital response data.
 */
void create_response(struct HTTP_REQUEST_STUCT *request, struct HTTP_RESPONSE_STRUCT *response){
    set_header_data(request, response);
    read_data(request, response);
    print_header(response);
}

/**
 * @brief - Open the file based on the filename in the request struct and copy the
 *          file data to the response struct.
 *
 * @param request - HTTP_REQUEST_STRUCT pointer containing filepath for requested resource.
 * @param response - HTTP_RESPONSE_STRUCT pointer where file data should be copied.
 */
void read_data(struct HTTP_REQUEST_STUCT *request, struct HTTP_RESPONSE_STRUCT *response){
    //Open the requested file in read-binary mode.
    FILE *fp = fopen(request->filepath, "rb");

    //Allocate dyamic memory based on conetent-length and copy file data.
    if(fp){
        response->data = (uint8_t*)malloc((*response).header.content_length + 1);
        fread(response->data, (*response).header.content_length, 1, fp);
        fclose(fp);
    }
}

/**
 * @brief - Set the header data struct with the date, server name, content-type and
 *          content-length
```

```
 *
 * @param request - HTTP_REQUEST_STRUCT pointer containing filepath for requested resource.
 * @param response - HTTP_RESPONSE_STRUCT pointer where header data should be set.
 */
void set_header_data(struct HTTP_REQUEST_STUCT *request, struct HTTP_RESPONSE_STRUCT *response){
    //Set the date that the file was accessed.
    set_date(&(response->header));

    //Set the server name
    set_hostname(&(response->header));

    //Set the content type of the response based on the filename extension
    set_content_type(request->filepath, &(response->header));

    //Set the content length based on the length of the file
    set_content_len(request->filepath, &(response->header));
}

/**
 * @brief - Converts the response header struct into a string.
 *
 * @param response - HTTP_RESPONSE_STRUCT pointer containing header data.
 */
void print_header(struct HTTP_RESPONSE_STRUCT* response){
    char header[1024], message[64];
    get_message((*response).header.status, message);

    //print HTTP version and response code/message
    strcpy(header, "HTTP/1.1 ");
    strcat(header, message);
    strcat(header, "\n");

    //print date data
    strcat(header, "Date: ");
    strcat(header, (*response).header.date);
    strcat(header, "\n");

    //print server name
    strcat(header, "Server: ");
    strcat(header, (*response).header.server);
    strcat(header, "\n");

    //print requested file's content type
    strcat(header, "Content-Type: ");
    strcat(header, (*response).header.content_type);
    strcat(header, "\n");

    //print requested file's length
    char length[10];
    sprintf(length, "%d", (*response).header.content_length);
    strcat(header, "Content-Length: ");
    strcat(header, length);
```

```
313         strcat(header, "\n");
314 
315         //print connection close
316         strcat(header, "Connection: close\n\n\0");
317 
318         //copy assembed string to response struct
319         response->header_str = (uint8_t*)malloc(strlen(header) + 1);
320         strcpy((char*)response->header_str, header);
321     }
322 
323     /**
324      * @brief - convert an int status into a string status message.
325      *
326      * @param status - int status code for response
327      * @param dest - string to copy message to
328      */
329     void get_message(int status, char * dest) {
330         switch(status) {
331             case 200:
332                 strcpy(dest, "200 OK\0");
333                 break;
334             case 404:
335                 strcpy(dest, "404 NOT FOUND\0");
336                 break;
337             case 405:
338                 strcpy(dest, "405 METHOD NOT ALLOWED\0");
339                 break;
340             case 418:
341                 strcpy(dest, "418 I'M A TEAPOT\0");
342                 break;
343             case 500:
344             default:
345                 strcpy(dest, "500 INTERNAL SERVER ERROR\0");
346                 break;
347         }
348     }
349 
350     /**
351      * @brief  - sets the content type header data by parsing the file extension
352      *           for the requested document.
353      *
354      * @param filename
355      * @param header
356      */
357     void set_content_type(char* filename, struct HTTP_RESPONSE_HEADER* header){
358         //Get substring after the final '.' character
359         char *extension = strrchr(filename, '.');
360 
361         if(strcmp(extension, ".html") == 0){
362             strcpy((*header).content_type, "text/html");
363         }else if(strcmp(extension, ".ico") == 0){
364             strcpy((*header).content_type, "image/x-icon");
```

```
365         }else if(strcmp(extension, ".jpg") == 0){
366             strcpy((*header).content_type, "image/jpeg");
367         }else if(strcmp(extension, ".txt") == 0){
368             strcpy((*header).content_type, "text/plain");
369         }else if(strcmp(extension, ".png") == 0){
370             strcpy((*header).content_type, "image/png");
371         }
372     }
373 
374     /**
375      * @brief - Set the content length field in the response header struct
376      *
377      * @param filepath - path of requested file
378      * @param header - header struct contained in http response struct.
379      */
380     void set_content_len(char* filepath, struct HTTP_RESPONSE_HEADER* header){
381         //Open file in read-binary mode
382         FILE* fp = fopen (filepath,"rb");
383 
384         if(fp){
385             //Get length of using file seeks
386             fseek(fp, 0, SEEK_END);
387             header->content_length = ftell(fp);
388             fseek(fp, 0, SEEK_SET);
389 
390             fclose(fp);
391         }
392     }
393 
394     /**
395      * @brief - set the date field in the http response header struct
396      *          by retrieving the current date using the time library.
397      *
398      * @param header - header struct contained in http response struct.
399      */
400     void set_date(struct HTTP_RESPONSE_HEADER* header){
401         time_t now = time(NULL);
402 
403         //Convert time to UTC format.
404         struct tm *t = gmtime(&now);
405 
406         if(t == NULL){
407             perror("Failed to obtain time");
408         }
409 
410         if(!strftime(header->date, sizeof(header->date)-1, "%a, %d %b %y %T %z", t)){
411             perror("Failed to format time and write to header");
412         }
413     }
414 
415     /**
416      * @brief - set the server field in the http_response header struct to the
```

```
417     *          hostname of this computer.
418     *
419     * @param header - header struct contained in http response struct.
420     */
421    void set_hostname(struct HTTP_RESPONSE_HEADER* header){
422        gethostname(header->server, sizeof(header->server));
423    }
424
425    ////////////////////////////////////////////////////////////////////////////////////////////////////
426    ////////////////////////////////////////////////////////////////////////////////////////////////////
427    ////////////////////////////////////////////////////////////////////////////////////////////////////
428    /**
429     * @file    - http_server.h
430     * @author  - Mitchell Larson (larsonma@msoe.edu)
431     * @brief   - Provides interface for an HTTP server.
432     * @version - 0.1
433     * @date    - 2019-01-17
434     *
435     * Copyright (c) 2019
436     *
437     */
438
439    #ifndef HTTP_SERVER
440    #define HTTP_SERVER
441
442    #include <stdio.h>
443    #include <unistd.h>
444    #include <stdlib.h>
445    #include <string.h>
446
447    #include "http_response.h"
448    #include "http_request.h"
449
450    /**
451     * @brief - Serves an HTTP client by reading and writing to a TCP socket
452     *
453     * @param client_sock_fd - TCP socket file descriptor
454     */
455    extern void serve_client(int client_sock_fd);
456
457    #endif
458
459    ////////////////////////////////////////////////////////////////////////////////////////////////////
460    ////////////////////////////////////////////////////////////////////////////////////////////////////
461    ////////////////////////////////////////////////////////////////////////////////////////////////////
462    /**
463     * @file     - http_response.h
464     * @author   - Mitchell Larson (larsonma@msoe.edu)
465     * @brief    - Interface and common struct for http response
466     * @version  - 0.1
467     * @date     - 2019-01-17
468     *
```

```
 * Copyright (c) 2019
 *
 */

#ifndef HTTP_RESPONSE
#define HTTP_RESPONSE

#include <stdio.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>

#include "http_request.h"

#define DATE_LEN 40

struct HTTP_RESPONSE_HEADER {
    int status;
    char date[40];
    char content_type[100];
    char server[256];
    int content_length;
};

struct HTTP_RESPONSE_STRUCT {
    struct HTTP_RESPONSE_HEADER header;
    uint8_t *data;
    uint8_t *header_str;
};

/**
 * @brief  - Create a HTTP response by setting header data and file data.
 *
 * @param request - HTTP_REQUEST_STRUCT pointer containing request data.
 * @param response - HTTP_RESPONSE_STRUCT pointer containing parital response data.
 */
extern void create_response(struct HTTP_REQUEST_STUCT *request, struct HTTP_RESPONSE_STRUCT *response);

#endif

////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////
/**
 * @file    - http_request.h
 * @author  - Mitchell Larson (larsonma@msoe.edu)
 * @brief   - Common struct for http request data.
 * @version - 0.1
 * @date    - 2019-01-17
 *
```

```
 * Copyright (c) 2019
 *
 */

#ifndef HTTP_REQUEST
#define HTTP_REQUEST

#include <stdio.h>
#include <unistd.h>
#include <stdint.h>

#define DATE_LEN 40

struct HTTP_REQUEST_HEADER {
    char HTTP_verb[7+1];
    char url[100];
    char version[10];
};

struct HTTP_REQUEST_STUCT {
    struct HTTP_REQUEST_HEADER header;
    char filepath[110];
};

#endif
```