```c
/*****************************************************************************
 * HTTP Server
 * CPE 3300, Daniel Nimsgern
 *
 * This HTTP server uses the machines port 80 to serve a HTML web page that
 * utilizes text images and audio. Once the client returns the requested values
 * the server saves those values and serves a new text and audio HTML page.
 *
 * I had many challenges thought the development many around where to start
 * when parsing the the HTTP request but with the help of stack overflow and
 * Zach I was able to come up with a method that works. I then did debugging
 * using valgrind and Microsoft CoPilot which was able to easily read my files
 * and tell me exactly what was wrong and give a suggestion on how to fix it.
 * almost every fix it suggested needed some tweaking to fit more with how I
 * wanted the program to operate but I was pleasantly surprised how well
 * CoPilot worked as a debugging assistant.
 *
 * Build with gcc -o httpwebserver httpwebserver.c
 *****************************************************************************/

/*============================================================================
  |          Includes
  ============================================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <bits/getopt_core.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>

/*============================================================================
  |          Global Variables
  ============================================================================*/

/* HTTP definitions */
#define DEFAULT_HTTP_PORT   (int)   80
#define HTTP_MAX_MSG_SIZE   (int)   1000000
#define HTTP_BASE_PATH          (char*) "./httpfiles"
#define MAX_PATH_LENGTH         (int)   1000
#define GET_REQUEST                     (char*) "GET"
#define PUT_REQUEST                     (char*) "PUT"

/* CLI ESC Codes */
#define ESC_BLACK_TXT       (char*) "\033[1;30m"
#define ESC_RED_TXT         (char*) "\033[1;31m"
#define ESC_GREEN_TXT       (char*) "\033[1;32m"
#define ESC_YELLOW_TXT      (char*) "\033[1;33m"
#define ESC_BLUE_TXT        (char*) "\033[1;34m"
#define ESC_MAGENTA_TXT     (char*) "\033[1;35m"
#define ESC_CYAN_TXT        (char*) "\033[1;36m"
#define ESC_WHITE_TXT       (char*) "\033[1;37m"
#define ESC_BR_GRAY_TXT     (char*) "\033[1;90m"
#define ESC_BR_RED_TXT      (char*) "\033[1;91m"
#define ESC_BR_GREEN_TXT    (char*) "\033[1;92m"
#define ESC_BR_YELLOW_TXT   (char*) "\033[1;93m"
#define ESC_BR_BLUE_TXT     (char*) "\033[1;94m"
#define ESC_BR_MAGENTA_TXT  (char*) "\033[1;95m"
#define ESC_BR_CYAN_TXT     (char*) "\033[1;96m"
```



```c
#define ESC_BR_WHITE_TXT    (char*) "\033[1;97m"

/*============================================================================
  |          Function Definitions
  ============================================================================*/

/**
 * @brief Main process - Handles HTTP requests to the server from clients
 *
 * @param argc Number of server configuration arguments
 * @param argv Array of server configuration arguments
 * @return int Program exit value
 */
int main(int argc, char** argv) {

        // Local Variables
        unsigned short port = DEFAULT_HTTP_PORT; // default port
        int sock; // socket descriptor

    // User argument parsing
    int c;
    while((c = getopt(argc,argv,"p:h"))!=-1)
    {
                switch(c)
                {
                        case 'p':
                                port = atof(optarg);
                                break;
                        case 'h':
                                printf("\n");
                printf("-h prints this help statement\n\n");
                printf("-p override the HTTP server port (default: 80)\n\n");
                                exit(1);
                                break;
                }
        }

        // ready to go
        printf("HTTP server over TCP configuring on port: %d\n",port);

        // for TCP, we want IP protocol domain (PF_INET)
        // and TCP transport type (SOCK_STREAM)
        // no alternate protocol - 0, since we have already specified IP

        if ((sock = socket( PF_INET, SOCK_STREAM, 0 )) < 0)
        {
                perror("Error on socket creation");
                exit(1);
        }

        // lose the pesky "Address already in use" error message
        int yes = 1;

        if (setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1)
        {
                perror("setsockopt");
                exit(1);
        }

        // establish address - this is the server and will
        // only be listening on the specified port
        struct sockaddr_in sock_address;
```

```
        // address family is AF_INET
        // our IP address is INADDR_ANY (any of our IP addresses)
    // the port number is per default or option above

        sock_address.sin_family = AF_INET;
        sock_address.sin_addr.s_addr = htonl(INADDR_ANY);
        sock_address.sin_port = htons(port);

        // we must now bind the socket descriptor to the address info
        if (bind(sock, (struct sockaddr *) &sock_address, sizeof(sock_address))<
0)
        {
                perror("Problem binding");
                exit(-1);
        }

        // extra step to TCP - listen on the port for a connection
        // willing to queue 5 connection requests
        if ( listen(sock, 5) < 0 )
        {
                perror("Error calling listen()");
                exit(-1);
        }

        // go into forever loop and echo whatever message is received
        // to console and back to source
        char* buffer = calloc(HTTP_MAX_MSG_SIZE*3, sizeof(char));
        char* command = calloc(HTTP_MAX_MSG_SIZE, sizeof(char));
        char* query = calloc(HTTP_MAX_MSG_SIZE, sizeof(char));
        char* version = calloc(HTTP_MAX_MSG_SIZE, sizeof(char));
        struct sockaddr_in callingDevice;
        socklen_t callingDevice_len;
        FILE* file;
    int fileLen;
        int bytesRead;
    int bytesWritten;
        int connection;

    while (1) {
                // hang in accept and wait for connection
                printf("====Waiting====\n");
                connection = accept(sock, (struct sockaddr*)&callingDevice,
                                &callingDevice_len);
                if (connection < 0)
                {
                        perror("ERROR on accept");
                }

        // Fork on connection to allow for multiple client connection
                int pid = fork();
                if (pid < 0)
                {
                        perror("ERROR on fork");
                }

        // Child Process
                if (pid == 0)
                {

            close(sock);
                        // ready to r/w - another loop - it will be broken when
```



```
the
            // connection is closed
                        while(1)
                        {
                                char* filePath = calloc(MAX_PATH_LENGTH, sizeof(
char));
                                char* response = calloc(HTTP_MAX_MSG_SIZE*10, si
zeof(char));
                                int incParams = 0;
                                int responseLen;

                                // read message
                                bytesRead = read(connection, buffer, HTTP_MAX_MS
G_SIZE-1);

                                if (bytesRead == 0)
                                {       // socket closed
                                        printf("====Client Disconnected====\n");
                                        close(connection);
                                        free(buffer);
                                        free(command);
                                        free(query);
                                        free(version);
                                        free(filePath);
                                        free(response);
                                        break;  // break the inner while loop
                                }

                                // print info to console
                                printf("Received HTTP request\n");

                                // Parse request
                                sscanf(buffer, "%s %s %s", command, query, versio
n);
                                printf("%s %s %s\n", command, query, version);

                                if (strchr(query, '?') != NULL)
                                {
                                        incParams = 1;
                                        printf("detected params\n");
                                }

                                char path[1000];
                                char rawparam[1000];
                                char* param;

                                if (incParams)
                                {
                                        char* tempPath = strtok(query, "?");
                                        strcpy(path, tempPath);
                                        printf("Path: %s\n", path);
                                        char* tempRawparam = strtok(NULL, "?");
                                        strcpy(rawparam, tempRawparam);
                                        printf("Parameters: %s\n", rawparam);
                                }
                                else
                                {
                                        strcpy(path, query);
                                        printf("%s\n",path);
                                }

                                if (incParams)
```

```
{
        FILE* csv = fopen("./httpfiles/cards.csv", "a");
        if (csv == NULL)
        {
                perror("Error opening file!\n");
        }

        param = strtok(rawparam, "&");
        while (param != NULL)
        {
                char name[HTTP_MAX_MSG_SIZE], va
lue[HTTP_MAX_MSG_SIZE];
                if(sscanf(param, "%[^=]=%s", nam
e, value) == 2)
                {
                        printf("Parameter: %s = %s\n"
, name, value);
                        fprintf(csv, "%s,", valu
e);
                }
                else
                {
                        printf("Failed to parse paramet
er: %s\n", param);
                }
                param = strtok(NULL, "&");
        }
        fprintf(csv, "\n");
        fclose(csv);
}

// replace + with spaces
for (int i = 0; i < strlen(path); i++)
{
        /* replace any + and % with space */
        if (path[i] == '+')
        {
                /* replace with space */
                path[i] = ' ';
        }

}

// Get file
if (!strcmp(command, GET_REQUEST))
{
        printf("file path preformatted: %s\n", path);
        if (strcmp(path, "/") == 0)
        {
                strcpy(path, "/index.html");
        }
}
strcat(filePath, HTTP_BASE_PATH);
strcat(filePath, path);

if (!strcmp(command, GET_REQUEST))
{
        /* Requesting file */
        printf("file path: %s\n", filePath);
        file = fopen(filePath, "r");

        if (file)
```



```
        {
                /* valid file */
                char* ext = strrchr(filePath, '.
');

                fseek(file, 0L, SEEK_END);
                fileLen = ftell(file);
                rewind(file);

                snprintf(response, HTTP_MAX_MSG_
SIZE*10,
                "%s 200 OK\r\nAccept-Ranges: bytes\r\nCont
ent-Type: %s"
                "\r\nContent-Length: %d\r\n\r\n",
                version, ext + 1, fileLen);

                responseLen = strlen(response);

                size_t filebytesread = fread(res
ponse + responseLen, 1,
                                                fil
eLen, file);
                responseLen += filebytesread;

                fclose(file);
        }
        else
        {
                /* invalid file */
                snprintf(response, HTTP_MAX_MSG_
SIZE*6,
                "%s 404 File Not Found",
                version);
        }
        printf("%s\n", response);
}

// send data to HTTP client
if ((bytesWritten = write(connection,
                          response,
  responseLen)) < 0)
{
        perror("Error sending file");
        exit(-1);
}
else
{
        printf("Bytes sent: %d\n", bytesWritten);
}

free(response);
free(filePath);

        } // end of child inner-while
        close(sock);
        exit(0);
    }
else
{
    close(connection);
}
```

```
    }   // end of outer loop
        // Free request buffers
        free(buffer);
        free(command);
        free(query);
        free(version);

        // should never get here
        return(0);
}
```